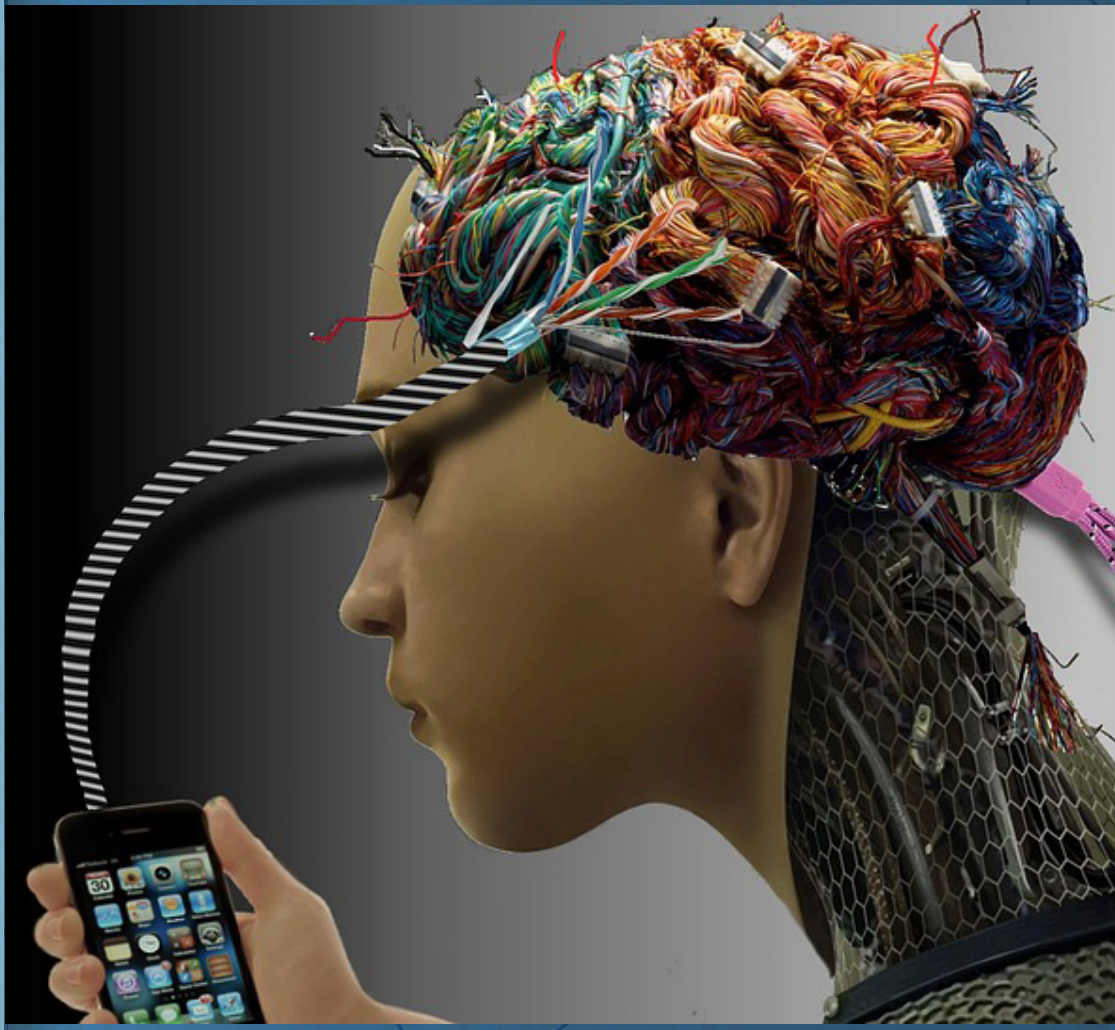
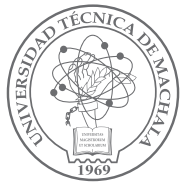
# Diseño de sistemas

Molina Ríos Jimmy, Valarezo Pardo Milton, Zea Ordoñez Mariuxi



Universidad Técnica de Machala

# Diseño de sistemas




Ing. César Quezada Abad, MBA
**Rector**

Ing. Amarilis Borja Herrera, Mg. Sc.
**Vicerrectora Académica**

Soc. Ramiro Ordóñez Morejón, Mg. Sc.
**Vicerrector Administrativo**

COORDINACIÓN EDITORIAL
VICERRECTORADO ACADÉMICO

Tomás Fontaines-Ruiz, PhD.
**Investigador Becario Prometeo-Utmach**
**Asesor Del Programa De Reingeniería**

Ing. Karina Lozano Zambrano
**Coordinadora Editorial**

Ing. Jorge Maza Córdova, Ms.
Ing. Cyndi Aguilar
**Equipo de Publicaciones**

# Diseño de sistemas

Jimmy Molina Rios
Milton Valarezo Pardo
Mariuxi Zea Ordoñez

UNIVERSIDAD TÉCNICA DE MACHALA
2015

Primera edición 2015

ISBN: 978-9942-24-075-0

D.R. © 2015, universidad técnica de machala
Ediciones utmach
Km. 5 1/2 Vía Machala Pasaje
www.utmachala.edu.ec

Este texto ha sido sometido a un proceso de evaluación por pares externos con base en la normativa editorial de la utmach.

Portada:
Concepto editorial
Samanta Cabezas (est. Comunicación Social)
Fotografia: Dir. de Comunicación UTMACH

Diseño, montaje y producción editorial: UTMACH

Impreso y hecho en Ecuador
*Printed and made in Ecuador*

Advertencia: "Se prohíbe la reproducción, el registro o la transmisión parcial o total de esta obra por cualquier sistema de recuperación de información, sea mecánico, fotoquímico, electrónico, magnético, electroóptico, por fotocopia o cualquier otro, existente o por existir, sin el permiso previo por escrito del titular de los derechos correspondientes".

# Índice

Prefacio ..... 17
Prólogo ..... 19
Dedicatoria ..... 21

CAPÍTULO 1 ..... 23
DEFINICIÓN DE LA ARQUITECTURA DEL SISTEMA ..... 23

1.1 Diseño arquitectónico. ..... 24
1.1.1 Decisiones de diseño arquitectónico. ..... 27
1.1.2 Organización del sistema. ..... 28
1.1.2.1 El modelo de repositorio de datos. ..... 28
1.1.2.2 El modelo cliente-servidor. ..... 30
1.1.2.3 El modelo de capas. ..... 31
1.1.3 Estilos de descomposición modular. ..... 32
1.1.3.1 Descomposición orientada a objetos. ..... 33
1.1.3.2 Descomposición orientada a flujos de funciones. ..... 34
1.1.4 Estilos de control. ..... 35
1.1.4.1 Control centralizado. ..... 36
1.1.4.2 Sistemas dirigidos por eventos. ..... 37
1.1.5 Arquitecturas de referencia. ..... 39
1.2 Arquitecturas de sistemas distribuidos. ..... 40
1.2.1 Arquitecturas multiprocesador. ..... 41
1.2.2 Arquitecturas cliente-servidor. ..... 42
1.2.3 Arquitecturas de objetos distribuidos. ..... 46
1.2.3.1 CORBA. ..... 48
1.2.4 Computación distribuida interorganizacional. ..... 50

1.2.4.1 Arquitectura peer-to-peer (p2p). ..... 51
1.2.4.2 Arquitectura de sistemas orientados a servicios. ..... 52
1.3 Arquitecturas de aplicaciones. ..... 54
1.3.1 Sistemas de procesamiento de datos. ..... 54
1.3.2 Sistemas de procesamiento de transacciones. ..... 56
1.3.2.1 Sistemas de información y de gestión de recursos. ... 57
1.3.3 Sistemas de procesamiento de eventos. ..... 60
1.3.4 Sistemas de procesamiento de lenguajes. ..... 61
CAPÍTULO 2 ..... 67
MODELADO DEL SISTEMA ..... 67

2.1 Diseño Orientado a Objetos. ..... 68
2.1.1 Objetos y clases ..... 70
2.1.1.1 Objetos concurrentes. ..... 74
2.1.2 Un proceso de diseño orientado a objetos. ..... 76
2.1.2.1 Contexto del sistema y modelos de utilización. ..... 79
2.1.2.2 Diseño de la arquitectura. ..... 81
2.1.2.3 Identificación de objetos. ..... 82
2.1.2.4 Modelos de diseño. ..... 85
2.1.2.5 Especificación de la interfaz de los objetos. ..... 90
2.1.3 Evolución del diseño ..... 90
2.2 Diseño de software de tiempo real. ..... 92
2.2.1 Diseño del sistema. ..... 94
2.2.1.1 Modelado de sistemas de tiempo real. ..... 96
2.2.2 Sistemas operativos de tiempo real. ..... 96
2.2.2.1 Gestión de procesos ..... 97
2.2.3 Sistemas de monitorización y control. ..... 99
2.2.4 Sistemas de adquisición de datos. ..... 101
2.3 Diseño de interfaces de usuario. ..... 107
2.3.1 Asuntos de diseño ..... 108
2.3.1.1 Interacción del usuario ..... 109
2.3.1.2 Presentación de la información ..... 118
2.3.2 El proceso de diseño de la interfaz de usuario ..... 119
2.3.3 Análisis del usuario. ..... 121
2.3.3.1 Técnicas de análisis ..... 123
2.3.4 Prototipado de la interfaz de usuario. ..... 125

2.3.5 Evaluación de la interfaz. ..... 132
3.1 Desarrollo rápido de software ..... 137
3.1.1 Métodos ágiles..... 140
3.1.2 Programación extrema..... 140
3.1.2.1 Pruebas en XP. ..... 143
3.1.2.2 Programación en parejas ..... 145
3.1.3 Desarrollo rápido de aplicaciones ..... 146
3.1.4 Prototipado del software ..... 150
3.2. Reutilización del software. ..... 152
3.2.1El campo de la reutilización..... 152
3.2.2 Patrones de diseño. ..... 154
3.2.3 Reutilización asentada en productores. ..... 156
3.2.4 Marcos de trabajo de aplicaciones. ..... 157
3.2.5 Reutilización de sistemas de aplicaciones. ..... 158
3.2.3.1 Reutilización de productos COTS..... 159
3.2.5.2 Líneas de productos software. ..... 162
CAPÍTULO 4..... 169
ESPECIFICACIONES PARA LA CONSTRUCCIÓN DE SISTEMAS ..... 169

4.1. Ingeniería del software basada en componentes..... 170
4.1.1 Componentes y modelos de componentes ..... 172
4.1.1.1 Modelos de componentes. ..... 174
4.1.1.2 Desarrollo de componentes para reutilización..... 174
4.1.2 El proceso CBSE. ..... 175
4.2 Desarrollo de sistemas críticos ..... 175
4.2.1 Métodos confiables ..... 180
4.2.2 Programación confiable. ..... 181
4.2.3 Tolerancia a defectos. ..... 188
4.2.3.1 Detección de efectos y evaluación de daños. ..... 189
4.2.3.2 Recuperación y reparación de defectos. ..... 193
4.2.4 Arquitectura tolerante a defectos. ..... 195
4.3 Evolución del software..... 199
4.3.1 Dinámica de evolución de los programas ..... 201
4.3.2 Mantenimiento del Software. ..... 202
4.3.2.1 Predicción del Mantenimiento..... 206

4.3.3 Procesos de evolución. ..... 209
4.3.3.1 Reingeniería de sistemas. ..... 212
4.3.4 Evolución del sistema heredado..... 217

Glosario ..... 227
Bibliografía ..... 229

# Índice de gráficos

Figura 1. Diagrama de bloques de un sistema robótico. ..... 26
Figura 2. Arquitectura de un conjunto de herramientas case integradas. ..... 29
Figura 3. Petición cliente – servidor. ..... 31
Figura 4. Modelo de capas. ..... 32
Figura 5. Modelo de objetos de un sistema de procesamiento de facturas. ..... 33
Figura 6. Modelo de objetos para la emisión de facturas al cliente. ..... 34
Figura 7. Modelo llamada-retorno. ..... 36
Figura 8. Manejador de eventos. ..... 37
Figura 9. Control dirigido por interrupciones. ..... 38
Figura 10. Modelo OSI. ..... 40
Figura 11. Sistema multiprocesador para el control de tráfico. ..... 42
Figura 12. Sistema cliente – servidor. ..... 43
Figura 13. Sistema de tres capas. ..... 43
Figura 14. Sistema de dos capas – Cliente ligero y pesado. ..... 44
Figura 15. Sistema bancario ATM. ..... 45
Figura 16. Arquitectura cliente – servidor de tres capas. ..... 45
Figura 17. Modelo cliente servidor de tres capas – Sistema bancario en internet. ..... 46
Figura 18. Arquitectura de objetos distribuidos. ..... 47
Figura 19. Estructura de una aplicación distribuida basada en CORBA. ..... 49

Figura 20. Comunicación de objetos a través de ORB. ..... 50
Figura 21. Arquitectura peer-to-peer descentralizada. ..... 51
Figura 22. Arquitectura peer-to-peer semicentralizada ..... 52
Figura 23. Componentes del procesamiento por lotes. ..... 55
Figura 24. Estructura de aplicaciones del
procesamiento de transacciones. ..... 56
Figura 25. Sistema de transacciones bancarias usando un
ATM. ..... 57
Figura 26. Middleware para gestión de transacciones. ..... 57
Figura 27. Sistema de información. ..... 58
Figura 28. Sistema bibliotecario LYBYS. ..... 58
Figura 29. Capas de un sistema de asignación de recursos. 59
Figura 30. Sistema multicapa de procesamiento de
transacciones por Internet. ..... 60
Figura 31. Modelo arquitectónico de un sistema edición. ..... 61
Figura 32. Arquitectura de un sistema de
procesamiento de lenguajes..... 62
Figura 33. Un modelo de flujo de datos de un compilador. ... 63
Figura 34. El modelo de repositorio de un sistema de
procesamiento de lenguajes..... 63
Figura 35. Un Sistema compuesto de objetos que interactúan
entre sí. ..... 69
Figura 36. Un objeto empleado. ..... 72
Figura 37. Un ejemplo de jerarquía con generalización. ..... 73
Figura 38. Un ejemplo objetos concurrentes. ..... 74
Figura 39. Implementación de un objeto utilizando
subprocesos de Java. ..... 76
Figura 40. Arquitectura de capas para un sistema de
creación de mapas meteorológico. ..... 78
Figura 41. Subsistemas en el ejemplo de mapas
meteorológicos. ..... 79
Figura 42. Casos de usos de la estación meteorológica. ..... 80
Figura 43. Un ejemplo de caso de uso “Informar”. ..... 81
Figura 44. La arquitectura de una estación meteorológica. .. 82
Figura 45. Ejemplos de clases en el sistema de
estación meteorológica ..... 84

Figura 46. Subsistema de una estación meteorológica. .... ....87
Figura 47. Secuencia de las operaciones de recogida de datos..... 89
Figura 48. Diagrama de estado para WeatherStation. ..... 89
Figura 49. Descripción de Java de la interfaz de la estación meteorológica ..... 91
Figura 50. Nuevos objetos para ayudar a la supervisión de la contaminación. ..... 91
Figura 51. Modelo general de un sistema en tiempo real..... 93
Figura 52. Procesos de control sensor/actuador. ..... 93
Figura 53. Modelo de máquina de estados de una bomba de petróleo..... 97
Figura 54. Múltiples interfaces de usuario..... 110
Figura 55. Presentación de la información..... 110
Figura 56. El modelo MVC de interacción del usuario..... 111
Figura 57. Presentaciones alternativas de la información. . 113
Figura 58. Métodos de presentar información numérica que varía dinámicamente ..... 113
Figura 59. Vista de Información gráfica que muestra valores relativos ..... 115
Figura 60. Un recuadro de entrada de texto utilizado por una asistente ..... 118
Figura 61. Mensajes de error al sistema y al usuario ..... 118
Figura 62. Un proceso de desarrollo iterativo..... 134
Figura 63. Desarrollo y prototipado incremental..... 137
Figura 64. El ciclo de entrega en la programación extrema 141
Figura 65. Tarjeta de historia para la descarga de documentos. ..... 142
Figura 66. Tarjetas de tareas para la descarga de documentos. ..... 144
Figura 67. Validez de la tarjeta de crédito. ..... 144
Figura 68. Un entorno de desarrollo rápido de aplicaciones ..... 146
Figura 69. Programación visual con reutilización..... 148
Figura 70. Vinculación de aplicaciones..... 149
Figura 71. Pruebas Back-to-back. ..... 151

Figura 72. Proceso de desarrollo de prototipos..... 151
Figura 73. Proceso de desarrollo de prototipos..... 173
Figura 74. Costes crecientes de la eliminación de defectos residuales..... 179
Figura 75. Excepción para manejar los números ingresados en una división. ..... 185
Figura 76. Excepción para el manejo de un numero ingresado dentro de un intervalo..... 187
Figura 77. Restricciones de los estados que se aplican en una bomba de insulina. ..... 190
Figura 78. Programa que calcula el mayor de tres números enteros en Java..... 191
Figura 79. Interfaz para ocupaciones de justificación o comprobación en Java..... 191
Figura 80. Procedimiento de ordenación seguro con recuperación hacia atrás..... 193
Figura 81. Programación con n-versiones. ..... 196
Figura 82. Redundancia modular Triple para tratar los fallos de ejecución del hardware. ..... 197
Figura 82. Bloques de Recuperación..... 197
Figura 84. Un modelo de desarrollo de software. ..... 200
Figura 85. Distribución del esfuerzo de mantenimiento. .... 204
Figura 86. Conste de desarrollo y mantenimiento. ..... 205
Figura 87. Predicción de mantenimiento..... 207
Figura 88. Identificación de los cambios y procesos de evolución. ..... 210
Figura 89. Implementación de Cambios..... 211
Figura 90. Proceso de reparación de emergencia..... 212
Figura 91. Ingeniería hacia adelante y reingeniería..... 214
Figura 92. El proceso de reingeniería. ..... 216
Figura 93. Aproximación de reingeniería. ..... 216
Figura 94. Evaluación de sistemas heredados ..... 2019

# Índice de tablas

Tabla 1. Principios de diseño de las interfaces de usuario.. 105
Tabla 2. Factores que implican un mensaje de error. ..... 117
Tabla 3. Los principios de los métodos agiles..... 139
Tabla 4. Prácticas de la programación extrema..... 139
Tabla 5. Características de los componentes. ..... 173
Tabla 6. Factores utilizados en la evolución del entorno. ... 222
Tabla 7. Factores Utilizados en la Evolución de la Aplicación. ..... 224

# Prefacio

Por medio del epígrafe de la Ingeniería del software se puntualizan una enorme cantidad de disciplinas, técnicas y metodologías que se referencian a todas las actividades relacionadas con la fabricación del software y su gestión, expuestas desde el enfoque de la ingeniería. Este concepto abre un estudio muy amplio que es totalmente nuevo para el alumno.

El éxito del desarrollo de un software es cuando satisfacen las necesidades de las personas que lo manipulan; en caso que funcione impecablemente durante mucho tiempo; en el momento que es fácil de modificar o incluso es más fácil de utilizar puede cambiar todas las cosas y de hecho las cambia para mejor. En el momento que falla un software es cuando los usuarios no se quedan satisfechos, cuando es propenso a errores; además, cuando es difícil de cambiar e incluso más difícil de utilizar, ocurren verdaderos desastres. Todos los programadores tienen la aspiración de desarrollar un software que haga bien las cosas, evitando que esas cosas malas ronden por las sombras de los esfuerzos fracasados. Para tener éxito al diseñar y construir un software necesitaremos disciplina. Es decir, necesitaremos un enfoque más allá de la ingeniería de sistemas.

Hoy en día, el software es una parte integral de la mayoría de los sistemas. Para ejecutar proyectos software de forma satisfactoria y construir productos de alta calidad, los profesionales del software necesitan entender las características únicas del software y el enfoque usado para desarrollar y mantener software. Este curso permitirá entender qué es el software y cuáles son los objetivos y componentes de la ingeniería del software, así como entender los conceptos de ciclo

de vida del software y metodología. Además, se verán los principales modelos de ciclo de vida del software y la diferenciación entre metodologías tradicionales y ágiles.

Se presentan los conceptos clásicos relacionados con el software y la Ingeniería del Software. El objetivo de este tema es tomar conciencia de la importancia de abordar la construcción del software desde una perspectiva de ingeniería. Se exponen los elementos constituyentes de un paradigma de desarrollo del software. Se ofrece una visión general del concepto de proceso, así como de los diferentes modelos de proceso software.

# Prólogo

En este libro se describe una guía para aquellas personas que desean aprender sobre el Diseño de Sistemas vinculado a la Ingeniera de Software.

En la actualidad la tecnología ha logrado inmiscuirse en las diferentes ramas de la ciencia, realizando la automatización de procesos en las diversas empresas en las que hace uso de las mismas; para todo esto el software tiene que pasar por un ciclo de vida de desarrollo; es donde la ingeniería de software y el diseño de sistemas intervienen.

En los últimos quince años la tecnología ha tenido un auge notable, y los cambios que se han encontrado en cada software se volvieron inimaginables. Las personas que se encargaron de crear nuevos sistemas indistintamente de la Ingeniería de Sistemas han mejorado inimaginablemente. Los servicios y edificaciones como son las centrales eléctricas, hidroeléctricas, sistemas de gestión y administración, telecomunicaciones, sistemas de guías de transporte. Necesitan obligadamente un sistema informático que sea eficaz, eficiente, íntegro, entre otros. Para poder realizar todos estos sistemas ya sean de mediana o gran estructura se han tenido que mezclar nuevas tecnologías y tecnologías anteriores pero que han mejorado notablemente sus versiones estas son: .NET, .CPP, .JAVA, .J2EE, acceden a nuevas aplicaciones que pueden estar basadas en Web o aplicaciones de escritorio. Pero hay que decir que en los últimos años tanto las aplicaciones Web y de escritorio han empezado a disminuir ya que han surgido las nuevas aplicaciones móviles, estas aplicaciones móviles lograron crecer a tal punto que muchas organizaciones antes de crear su página web crean primero la aplicación para un Smartphone.

No obstante surge un desconsuelo porque los métodos y técnicas de la ingeniería de software se han mantenido igual, un ejemplo de esto se logró reconocer que una de las metodólogas más usadas como es la metodología en cascada tenía varios problemas.

El uso de las herramientas CASE, siguen siendo usadas y basadas en UML y UML2, siguen ayudando a los diseñadores a crear diagramas que tengan la funcionalidad del sistema que se está creando, generar código y hasta un cierto punto crear ingeniería inversa.

Las metodologías del ciclo de desarrollo de software conjuntamente con sus técnicas lograron ayudar a la desarrollar sistemas complejos y de gran tamaño a ser mejores y cumplan con cada requerimiento que se necesita para solucionar un problema de la vida cotidiana. Los últimos avances se puede denotar que en la ingeniería en software ha sido la visión de UML que se incorporó como un gran estándar para la descripción de sistemas orientados a objetos, bajo el desarrollo de métodos agiles como es la metodología XP que utiliza una programación extrema.

Por ende, este texto no procura ser un texto que trata sobre todas las metodologías agiles si no asentar los procesos básicos de la ingeniería de software como son las especificaciones, diseño, implementación, verificación y validación conjuntamente con gestión. Por eso es de obligación de los stakeholders que entiendan todos los procesos y técnicas más adecuadas para adaptar a las situaciones particulares. En cada capítulo de este libro se intenta explicar las técnicas necesarias y específicas de la ingeniera del software que son necesarias para la construcción de los sistemas críticos.

Irremediablemente, todos los libros irradian las distintas opiniones de sus autores. Por eso es que muchos lectores se pondrán en disconformidad con respecto a las opiniones y deliberación del material. Sin embargo, se espera que todos los ingenieros de software y futuros ingenieros en la informática les resulten muy interesantes y propongan críticas constructivas a los temas aqui tratados..

# Dedicatoria

*Este libro es dedicado a Dios.*

# Definición de la arquitectura del sistema

A continuación se detalla el objetivo principal de cada contenido del capítulo 1, Definición de la arquitectura del sistema:

El objetivo general del tema 1.1. Diseño arquitectónico, es dar a conocer los conceptos y principios generales de la arquitectura del software y del diseño arquitectónico. A continuación se lista los temas a tratar dentro de este contenido:

- Decisiones de diseño arquitectónico.
- Organización del sistema.
- Estilos de descomposición modular.
- Estilos de control.
- Arquitecturas de referencia.

El objetivo general del punto 1.2. Arquitecturas de sistemas distribuidos, es conocer y aplicar los distintos modelos de arquitectura del software para los sistemas distribuidos. A continuación se lista los temas de importancia a tratar:

- Arquitecturas multiprocesador.
- Arquitecturas cliente-servidor.
- Arquitecturas de objetos distribuidos.
- Computación distribuida inter-organizacional.

El objetivo general del punto 1.3. Arquitecturas de aplicaciones, es conocer y aplicar los distintos modelos arquitectónicos para realizar clases específicas. A continuación se lista los temas de importancia a tratar:

- Sistemas de procesamiento de datos.
- Sistemas de procesamiento de transacciones.
- Sistemas de procesamiento de eventos.
- Sistemas de procesamiento de lenguajes.

### 1.1 Diseño arquitectónico.

Cada sistema, en especial los grandes, están compuestas por subsistemas. Cuando se empieza con el diseño estructural, se trata de identificar cuáles son estos subsistemas para establecer una medida de control y comunicación entre estos, esto es lo que se conoce como diseño arquitectónico. Este diseño arquitectónico nos brinda una descripción de las cualidades de la aplicación a desarrollar.
Para la realización de una buena arquitectura del software es necesario considerar como esencial las siguientes actividades:

- Charla con los stakeholders: La arquitectura del software puede ser discutido por el grupo de interés de la aplicación a desarrollar.
- Análisis del sistema: Se necesita realizar un previo análisis, el cual nos permitirá tomar decisiones en cuanto al diseño arquitectónico, logrando cumplir los requisitos más críticos como el rendimiento o la mantenibilidad de la aplicación a desarrollar.
- Reutilización a gran escala: En la mayoría de los casos, las arquitecturas de diferentes aplicaciones son las mismas, puesto que algunos sistemas tienen requerimientos parecidos y por lo tanto puede existir reutilización de estas a gran escala.

También se puede sugerir que la arquitectura de software sirva como una estrategia de diseño para establecer los requerimientos del sistema; también usarlo para negociar con el cliente, los programadores y desarrolladores, llegando a ser una herramienta muy importante para la gestión permitiendo a los diseñadores concentrarse en las abstracciones del sistema. La arquitectura del sistema tiene su lado negativo, puede afectar al sistema ya sea en su rendimiento, su solidez o dificultar la mantenibilidad del mismo.

La arquitectura del sistema puede depender de los requerimientos no funcionales obtenidos en el análisis: entre los que tienen mayor relevancia para juzgar la operación de un sistema tenemos:

1. **Rendimiento:** La arquitectura se debe diseñar para localizar las operaciones críticas en cada subsistema, se puede usar componentes de grano grueso en vez de granos finos para así disminuir la comunicación entre subsistemas.
2. **Protección:** La arquitectura debe utilizar una estructura en

capas, con los recursos más sensibles de cada capa interna aplicando una seguridad de alto nivel.

3. **Seguridad:** La arquitectura se debe diseñar de tal forma que todas las operaciones de seguridad estén en un único subsistema o en otro caso en un pequeño grupo de subsistemas. Esto se hace para reducir los costes y problemas de validación con la seguridad, logrando crear sistemas de protección.

4. **Disponibilidad:** La arquitectura se debe diseñar para incluir componentes redundantes de la aplicación, logrando reemplazar o actualizar estos sin detener el sistema.

5. **Mantenibilidad:** La arquitectura se debe diseñar usando componentes que sean independientes dentro del sistema, estos deben ser de grano fino para facilitar su modificación.

Pueden existir conflictos entre arquitecturas, citando un ejemplo, si se usa componentes de grano grueso el sistema mejora en rendimiento y si se usa componentes de grano fino mejora la mantenibilidad del mismo, ambas propiedades son importantes para la aplicación por lo que deben funcionar como un conjunto, esto se puede conseguir usando distintos estilos arquitectónicos para cada parte del sistema.

Se puede definir un diseño de un subsistema como aquella descomposición abstracta de un software, la cual está dividida en componentes de grano grueso, siendo estos importantes para sí mismos.

Se utiliza los diagramas de bloques para describir estos diseños de subsistemas en donde cada bloque representa a cada uno. Bloques dentro de otros bloques representan la descomposición de un subsistema, las flechas indican la comunicación entre subsistemas. Estos diagramas de bloques representan un diseño de alto nivel, por lo cual personas de diferentes carreras que estén incluidas dentro del proceso de desarrollo de la aplicación puedan entenderlos sin ningún inconveniente.

En la figura 1 se puede observar un modelo de diagrama de bloque para un sistema robótico, el cual representa los subsistemas que deben desarrollarse para realizar el respectivo control de empaquetado. Este robot puede empaquetar distintos objetos, para eso utiliza un subsistema que le permite tener una visión, identificarlos y elegir el tipo de empaquetado, después el sistema mueve los objetos seleccionados hacia una cubeta transportadora.

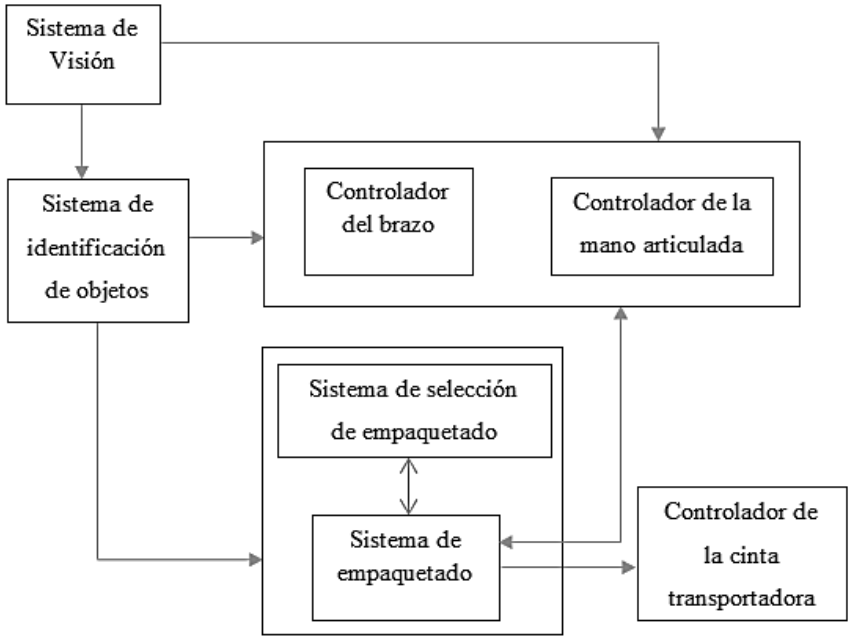



***Figura 1.*** Diagrama de bloques de un sistema robótico.

Para un diseñador de software este proceso es correcto, pero este tipo de modelo está enfocado a los stakeholders, ya que el grupo de interés de la aplicación a desarrollar logra tener una noción de cómo funciona el sistema, permitiendo identificar los subsistemas importantes y así asignar personas encargadas al desarrollo de los mismos de forma independiente. Este tipo de modelo no son los únicos que se utilizan para representaciones arquitectónicas pero es uno de los más usados y útiles por los diseñadores de software.

Es difícil decidir cómo descomponer un sistema en subsistemas pero siempre se debe crear un diseño que tenga relación entre los requerimientos y subsistemas de la aplicación a desarrollar, es decir, si los requerimientos en algún punto llegan a cambiar, este cambio solo afectará al subsistema involucrado y no a todo el sistema.

### 1.1.1. Decisiones de diseño arquitectónico.

Se puede definir el diseño arquitectónico como aquel proceso de mucha imaginación en la cual se quiere satisfacer los requerimientos funcionales y no funcionales de un sistema. Las actividades que se ven involucradas en el proceso del diseño arquitectónico cambian dependiendo del tipo de aplicación a desarrollar, también depende de la experiencia del desarrollador y de los requerimientos del sistema.

Entonces todo el proceso que conlleva el diseño arquitectónico se debe observar desde un punto de vista de decisión y no de actividades, ya que en cada proceso los diseñadores tienen que tomar decisiones importantes que van afectar el sistema y por ende a su proceso de desarrollo.

Cada sistema es diferente y único, pero cuando están relacionados en el mismo domino estas aplicaciones tienen arquitecturas similares, pudiendo ser genéricas. Mas adelante, estudiaremos a fondo ejemplos referentes a las aplicaciones genéricas.

Si se analiza los sistemas embebidos o aquellos sistemas diseñados para computadoras personales, estos por lo general utilizan un solo procesador por lo que no será necesario diseñar una arquitectura distribuida. Pero la mayoría de los sistemas actuales son sistemas distribuidos, ya que su software se encuentra distribuido en varias computadoras, entonces elegir la arquitectura de distribución correcta será algo decisivo para el rendimiento del sistema.

La arquitectura de un software puede basarse en algún estilo arquitectónico, se puede definir como un estilo arquitectónico a una serie de patrones de organización como por ejemplo el modelo cliente-servidor o por capas. La mayoría de las arquitecturas de los sistemas grandes no usan un único estilo, cada parte de la aplicación puede tener un estilo distinto, es decir pueden crear la arquitectura de un sistema con la combinación de diferentes estilos arquitectónicos.

La noción de Garlan y Shaw de un estilo arquitectónico incluye las tres siguientes cuestiones de diseño:

1. Elegir la estructura más adecuada, ya sea cliente-servidor o por capas, todo depende de la cantidad de requisitos que se logre cumplir en el sistema al utilizarla.
2. Descomponer el sistema en módulos o subsistemas, se debe

decidir cuál estrategia usar para realizar este proceso.

3. Al finalizar el proceso arquitectónico se debe realizar un entregable llamado documento de diseño arquitectónico, el cual incluye muchas representaciones gráficas del sistema con su respectiva estructura de módulos y los modelos implementados.

Estos modelos arquitectónicos pueden ser:

1. Un modelo estructural estático, en este modelo se muestran los componentes que han sido desarrollados por separado.
2. Un modelo de proceso dinámico, en este modelo se organiza el sistema en forma de procesos, se lo realiza conforme avance el desarrollo de la aplicación.
3. Un modelo de interfaz, en este modelo se define los servicios de cada subsistema utilizando una interfaz gráfica.
4. Modelos de relaciones, en este modelo se muestran todas las relaciones ya sea entre los subsistemas o los flujos de datos.
5. Un modelo de distribución, en este modelo de detalla cómo fueron distribuidos los subsistemas a las demás computadoras.

### 1.1.2. Organización del sistema.

La organización del sistema nos sirve para elegir la estrategia a utilizar en nuestra aplicación, con la finalidad de tomar decisiones sobre qué modelo organizacional utilizar al principio de cada proceso del diseño arquitectónico. Esta organización del sistema se ve reflejado en las estructuras de los subsistemas, pero es más frecuente que los subsistemas incorporen más detalles que el modelo organizacional.

La organización del sistema está conformada por tres estilos organizacionales. Todos estos modelos se pueden usar juntos o por separados.

1. Un modelo de repositorio de datos.
2. Un modelo cliente – servidor.
3. Un modelo de capas.

#### 1.1.2.1. El modelo de repositorio de datos.

Cada subsistema debe intercambiar información con otros subsistemas para que puedan trabajar normalmente y de forma efectiva, esto se lo logra mediante dos formas:

- Todos los datos de los usuarios se almacenan en una base de datos central y cada computadora puede acceder a esa infor-

mación.

- Cada subsistema tiene su propia base de datos lo que permite intercambiar información con los demás subsistemas.

Los sistemas que almacenan mucha información utilizan una base de datos compartida o repositorios de datos, por lo tanto este modelo es adecuado para aplicaciones donde cada dato de un subsistema es usado por otro, por ejemplo los sistemas de mando y control, los CAD, las herramientas case, entre otros. En la figura 2 se puede observar como una arquitectura de herramientas case utiliza el modelo de repositorios compartidos. Implementar este estilo organizacional genera algunas ventajas y desventajas:

- Tener un repositorio compartido nos ayuda a compartir de manera eficiente grandes cantidades de información.
- Puede haber un bajo rendimiento si los modelos utilizados en su sistema no se ajustan a los repositorios compartidos.
- Sera difícil cambiar de un modelo que no use un repositorio a uno que sí, generará muchos costos, hasta puede ser imposible.
- Los repositorios están encargados de las copias de seguridad, protección y el acceso a los datos.
- Este modelo impone su política de uso a todos los subsistemas.
- El modelo de compartición es visible a través del esquema del repositorio.
- Las nuevas herramientas se integran de forma directa puesto

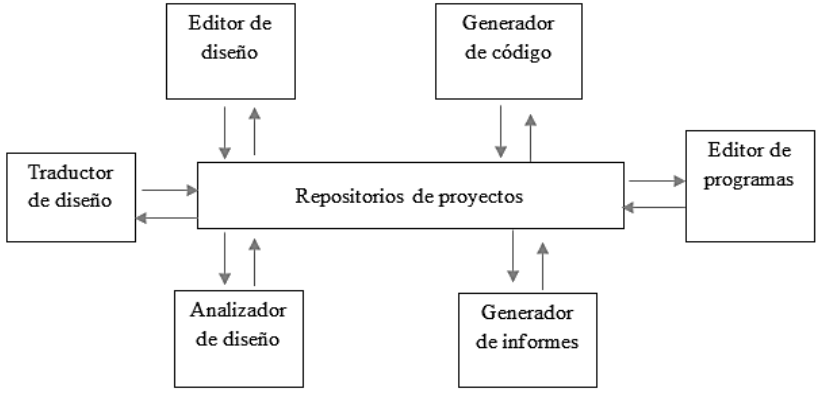



***Figura 2.*** Arquitectura de un conjunto de herramientas case integradas.

que éstas son compatibles con el modelo de datos acordado.

**1.1.2.2. El modelo cliente-servidor.**

Es un modelo arquitectónico donde el sistema está conformado por un conjunto de servidores y los clientes (usuarios) usan estos servicios. Entre los principales componentes que conforman el modelo cliente - servidor están:

1. Servidores.
2. Clientes.
3. Red.

Cada cliente de la red puede conocer que servidores están disponibles en la web y cuáles son los servicios que ofrecen, pero en cambio un servidor no necesita conocer que cliente accedió a esa información o cual es el volumen de clientes que están actualmente usando dicha información. Los clientes pueden ver estos datos desde la web usando el protocolo http.

En otras palabras, cuando un cliente realiza una petición a un servidor, este debe esperar hasta que reciba una respuesta, en la figura 3 se observa este comportamiento, es un sistema multiusuario que esta online, ofrece a sus clientes películas y fotos. En este sistema web hay algunos servidores que ayudan a gestionar el sistema.

Cada cliente de la red puede conocer que servidores están disponibles en la web y cuáles son los servicios que ofrecen, pero en cambio un servidor no necesita conocer que cliente accedió a esa información o cual es el volumen de clientes que están actualmente usando dicha información. Los clientes pueden ver estos datos desde la web usando el protocolo http.

En otras palabras, cuando un cliente realiza una petición a un servidor, este debe esperar hasta que reciba una respuesta, en la figura 3 se observa este comportamiento, es un sistema multiusuario que esta online, ofrece a sus clientes películas y fotos. En este sistema web hay algunos servidores que ayudan a gestionar el sistema.

La ventaja más importante que tiene el modelo cliente - servidor es que es considerada una arquitectura distribuida. Se puede tener sistemas en red y tener muchos procesadores distribuidos, de igual manera es muy fácil añadir nuevos servidores o actualizarlo. En el punto 1.2. Arquitecturas de sistemas distribuidos, se estudiará más detallada-

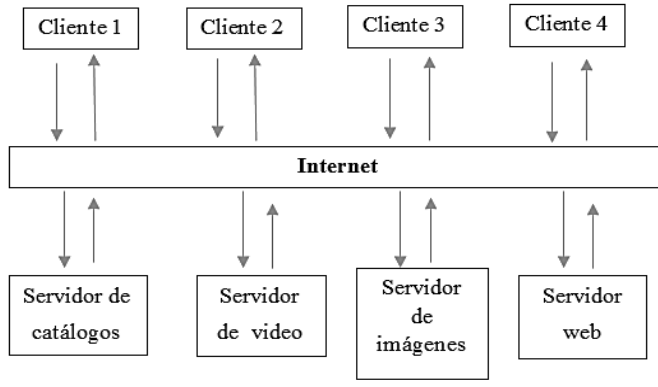



***Figura 3.*** Petición cliente – servidor.

mente su importancia.

**1.1.2.3. El modelo de capas.**

Este modelo organiza el sistema en capas y cada una proporciona una serie de servicios, por lo tanto cada capa puede identificarse como si fuera una máquina abstracta, donde el lenguaje de máquina se da por los servicios que proporcionan.

El modelo de capas se lo construye sobre un sistema de base de datos, lo que permitirá el almacenamiento de los datos y servicios como transacciones, además de la recuperación de información y el control de acceso. Las bases de datos usan las facilidades que ofrecen los sistemas operativos y almacenan ficheros para su implementación.

Este modelo es soportado por la metodología de desarrollo incremental, debido a que cada vez que se realiza una capa los servicios que esta brinda pueden estar funcionando para los clientes. Si una capa esta sin cambios, puede ser reemplaza por otra capa igual, además si una capa cambia o se le añade nuevas características solo se verá afectada la capa adyacente a ella.

La estructuración de este modelo puede ser difícil. Las capas internas nos proporcionan facilidades básicas, como gestionar ficheros, ya que cada nivel lo requiere. Como se observa en la figura 4, un usuario requiere un servicio de alto nivel, ya que tiene que pasar por todas las capas adyacentes para tener acceso a las capas inferiores, esto hace que el sistema sea lento.

En el caso de que existan varias capas, cuando una capa que se

encuentra en un nivel superior pida un servicio, este servicio tendrá que ser interpretado muchas veces en diferentes capas hasta ser procesado. Entonces para evitar este inconveniente, el sistema tiene que comunicarse directamente con las capas que se encuentran en un nivel más bajo, en lugar de utilizar aquellos servicios que ofrecen las

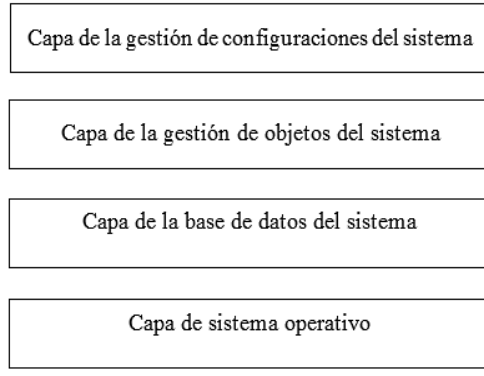



***Figura 4.*** Modelo de capas.

capas adyacentes.

**1.1.3. Estilos de descomposición modular.**

Una vez se haya elegido el tipo de organización del sistema, se procede a elegir la aproximación para descomponer cada sistema en módulos o subsistemas. Ya se han visto varios estilos en la sección 1.1.2, sin embargo, cada componente de estos módulos son más pequeños que en los subsistemas, lo cual utilizar la descomposición modular será la mejor opción.

Para no confundir el concepto de un subsistema y un módulo, se analizarán estos términos por separado:

- Un subsistema se lo define como otro sistema dentro de un mismo sistema, debido a que el funcionamiento de estos subsistemas no requiere de los servicios de otros, ya que cada uno tiene sus propios módulos e interfaces permitiendo la comunicación entre ellos.
- A un módulo lo podemos definir como aquel componente propio de un subsistema, el cual proporciona uno o varios servicios a otros módulos.

Para descomponer un subsistema en módulos debemos tener presente estas dos estrategias:

1. Descomposición orientada a objetos.
2. Descomposición orientada a flujos de funciones.

### 1.1.3.1.Descomposición orientada a objetos.

La figura 5 nos da un ejemplo de cómo es un modelo orientado a objetos de un sistema de procesamiento de facturas, este sistema puede emitir facturas a los clientes, recibir y emitir pagos a los proveedores. Se definen las operaciones o funciones antes mencionadas en la parte inferior del objeto denominado factura, las flechas indican que ese ob-

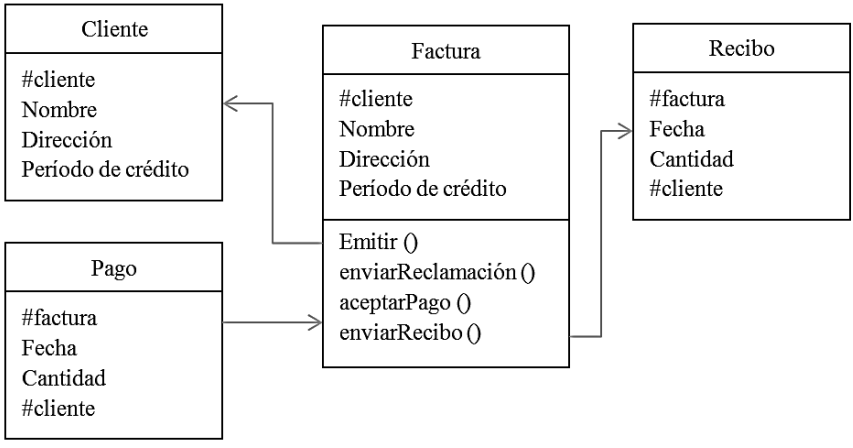



***Figura 5.*** Modelo de objetos de un sistema de procesamiento de facturas.

jeto utiliza atributos proporcionados por otro objeto.

Se puede definir a la descomposición orientada a objetos como aquella descomposición que tiene una relación con las clases, atributos y métodos. Los objetos se crean a partir de las clases, por ejemplo, se tiene una clase factura con varios métodos y esta clase a su vez utiliza otras clases como la de cliente, pagos y recibos para realizar un proceso de facturación.

Se puede implementar objetos sin afectar a otros objetos. Un objeto se lo define como una representación del mundo real, debido a que se lo utiliza una y otra vez; en la programación los objetos pueden

ser utilizados varias veces, para ello existen lenguajes de programación orientados a objetos que nos ofrece este componente arquitectónico.

**1.1.3.2.Descomposición orientada a flujos de funciones.**

En este tipo de descomposición se encuentran transformaciones, las cuales son funcionales, estas transformaciones procesan sus entradas y dan como resultado una salida, esto quiere decir que los datos van a fluir de una función a otra, y a medida que van avanzando se van a ir transformando. Entonces los datos de entrada van a fluir por medio de estas transformaciones hasta que dé como resultado un dato de salida. Estas transformaciones pueden ser realizadas ya sea en secuencia o en paralelo.

Este modelo se lo conoce como tubería o filtro, y se guía por la terminología del sistema operativo Linux, la cual permite conducir datos y comandos que vendrían a ser las transformaciones funcionales. Se usa la palabra filtro debido a que las transformaciones filtran todos los datos de entrada.

En la figura 6 observamos un ejemplo parecido a la descomposición anterior, aquí una empresa emite facturas a sus clientes. En cada semana se hace una comparación entre los pagos con las facturas, cada factura pagada tiene un recibo y para aquellas factura que no han recibido un pago dentro un plazo establecido se emite una reclamación. Este ejemplo es un modelo de una parte del sistema de facturas. La

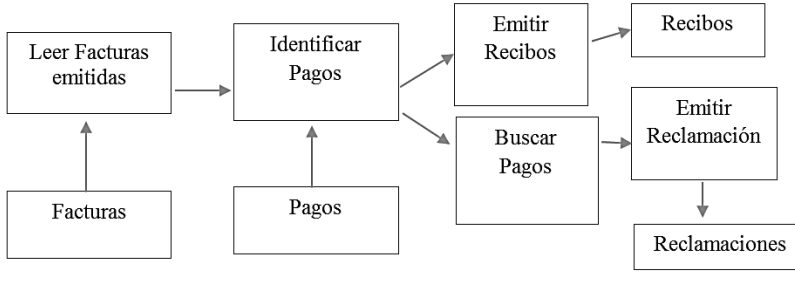



***Figura 6.*** Modelo de objetos para la emisión de facturas al cliente.

diferencia con el modelo de objetos es que este es más abstracto.
Utilizar este modelo brinda algunas ventajas:

1. Permite utilizar muchas veces las transformaciones.
2. Es intuitiva.
3. Se puede hacer evolucionar el sistema cada vez que se añade nuevas transformaciones.
4. Sencilla de implementar.

Este modelo tiene un problema, debe existir un formato base para que los datos transferidos puedan ser detectados por las demás transformaciones, es decir, que cada transformación debe ser igual con la que se está comunicando. Cada transformación debe seguir un formato estándar para que puedan comunicarse entre sí. Por ejemplo, en el sistema operativo Unix el formato que siguen estas transformaciones es solamente una cadena de caracteres. Como se mencionó anteriormente cada transformación procesa su entrada y da como resultado una salida, con el formato elegido para estas transformaciones. Pero esto puede ocasionar la sobrecarga del sistema impidiendo agregar nuevas transformaciones con formatos distintos.

Hay sistemas difíciles de expresar usando este modelo como por ejemplo los sistemas interactivos, ya que necesitan de un flujo de datos de entrada para procesar una salida. Pero hay otros sistemas que se pueden modelar usando esta arquitectura como pueden ser los modelos textual con entradas y salidas, también las interfaces gráficas con eventos del ratón o teclado.

### 1.1.4.Estilos de control.

En los puntos anteriores estudiamos como el modelo estructural nos ayuda a descomponer un sistema en subsistemas, pero cada subsistema debe ser controlado para que los servicios que envíe lleguen al lugar correcto y al momento dispuesto. Ya que los modelos estructurales no contienen un estilo de control, el diseñador debe elegir que subsistema lo tendrá.
Existen dos estilos de control:

1. Control centralizado.
2. Control basado en eventos.

**1.1.4.1.Control centralizado.**

En este modelo se selecciona un subsistema que se denominará controlador del sistema, este será el encargado de gestionar las tareas de los otros subsistemas. Estos modelos centralizados se dividen en dos clases:

- Modelo llamada-retorno: es un modelo de subrutina en forma de árbol descendente, su control comienza al iniciarse una subrutina y con cada llamada a otras subrutinas va bajando por los niveles del árbol, este tipo de modelo solo se aplica a sistemas secuenciales.
- Modelo gestor: a diferencia del modelo anterior, este modelo se lo utiliza en sistemas concurrentes. Un componente del sistema se diseña como un gestor que controla los inicios y paradas de los demás procesos de la aplicación.

En la figura 7 observamos un modelo llamada-retorno, en donde el programa principal puede contactar a cualquiera de sus tres rutinas, y estas a su vez pueden llamar a mas subrutinas, cabe recalcar que no es un modelo estructural.

Estas subrutinas en los lenguajes de programación son llamadas funciones, pero en los lenguajes orientados a objetos estas funciones se las conoce como métodos.

Este modelo permite entender con facilidad los flujos de control

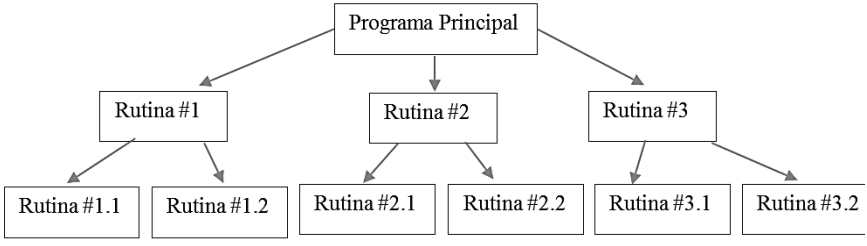



***Figura 7.*** Modelo llamada-retorno.

y conocer como el sistema va a responder a ciertas entradas, pero las excepciones que hay que controlar son un poco molestas.

Este modelo también se lo conoce como ciclo de eventos, debido a que el proceso controlador del sistema es el que toma la decisión de comenzar o finalizar un proceso. Para comprobar si un proceso ha

producido información el controlador generar ciclos una y otra vez para detectar eventos o cambios.

**1.1.4.2.Sistemas dirigidos por eventos.**

Estos modelos se guían por eventos que se generan externamente. El evento no solamente es una señal binaria, también puede estar dentro de un rango de valores. La diferencia entre un evento y una entrada es que el evento aparece fuera del control del proceso.

Se analizan dos modelos de control dirigidos por eventos:

1. Modelos de transmisión: Todo evento se transmite hacia los otros subsistemas. Estos modelos son efectivos para construir subsistemas que se encuentren distribuidos en una red.

2. Modelos dirigidos por interrupciones.: Las interrupciones que se den externamente son detectadas, esto se conoce como manejador de interrupciones, normalmente estos modelos se los usa en sistemas de tiempo real con requerimientos temporales rígidos.

Si se toma como ejemplo un modelo de transmisión, cada subsistema registra eventos específicos, si uno de estos eventos se pone en marcha, el control va a formar parte del subsistema que pueda controlar este evento. Una diferencia entre este modelo con el centralizado se lo muestra en la figura 8, la cual nos indica que la política de control no está impregnada al manejador de eventos. En otras palabras cada subsistema decide que evento necesita y la tarea del manejador de eventos es de enviarlos a los demás subsistemas.

Si se envía todos los eventos a todos los subsistemas, se puede generar una sobrecarga de procesamiento, frecuentemente el manejador de eventos realiza un registro de todos los subsistemas. Los subsistemas generan eventos para ver si algún dato está disponible y el manejador de eventos los localiza, los consulta y los envía a todos aquellos subsistemas que estén interesados por él. Estos manejadores de evento

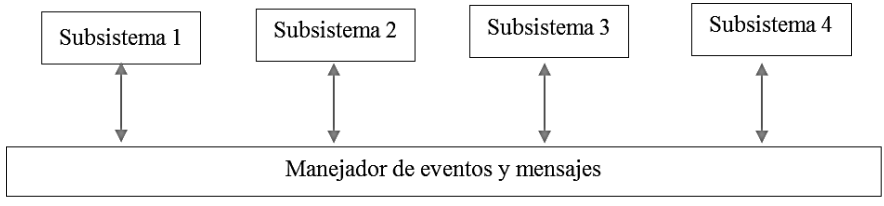



***Figura 8.*** Manejador de eventos.

también pueden soportar comunicaciones punto a punto.

La evolución de un modelo de transmisión es simple, ya que cada nuevo subsistema puede formar parte del sistema, manejar clases que controlen eventos y registrar sus sucesos en el manejador de eventos. Un subsistema puede poner en marcha a otro subsistema sin conocer información sobre este, permitiendo su ejecución en máquinas distribuidas.

Los subsistemas no saben cuáles son los eventos que van a controlar, ni en qué momento van a controlarlos, es decir, que un subsistema cuando genera un evento no tienen noción sobre que subsistema se va a registrar y esto ocasiona problemas en los resultados del manejador de eventos.

En la figura 9 se observa un control dirigido por interrupciones, cada interrupción está asociada a una memoria, la cual almacena la dirección del manejador. Al momento de recibir una interrupción, un conmutador hardware transfiere el control a este manejador, el cual puede iniciar o detener otros procesos como una respuesta al evento generado por la interrupción. En este modelo cada vez que se produce

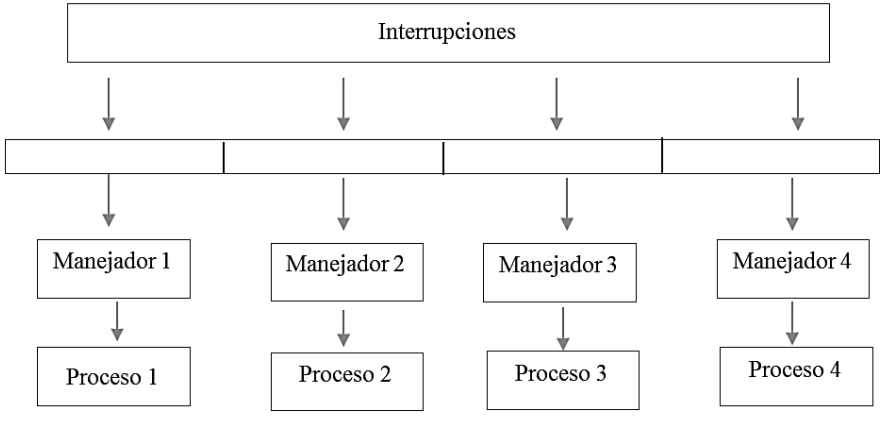



***Figura 9.*** Control dirigido por interrupciones.

un evento se necesita de una respuesta inmediata, por eso este modelo se lo usa principalmente en sistemas de tiempo real.

Este modelo nos brinda respuestas rápidas al momento de pro-

ducirse un evento pero la programación y sus validaciones son muy complejas, es difícil replicar patrones cuando se hace pruebas en el sistema o cambiar algún sistema desarrollado por este modelo cuando la cantidad de interrupciones se ve limitada por el hardware usado. Si las interrupciones alcanzan el límite del hardware, no funcionará ningún evento.

### 1.1.5. Arquitecturas de referencia.

Todos los modelos estudiados anteriormente se pueden aplicar a muchos sistemas. En las arquitecturas de referencia se encuentran las arquitecturas de dominio específico.

Entre los modelos arquitectónicos de dominio específico se encuentran:

1. Modelos genéricos: Estos modelos encapsulan características importantes de cada sistema y al ser abstracciones se pueden conseguir de uno o varios sistemas reales.
2. Modelos de referencia: Estos modelos son abstractos pero dan una descripción más amplia de una clase de sistema, permitiendo a los diseñadores conocer la estructura de estas. Estos modelos se obtienen de un estudio de domino realizado al sistema.

Se define al modelo OSI como un modelo que está conformado por siete capas que permiten interconectarse con otros sistemas abiertos. En la figura 10 se observa el modelo OSI, donde las capas más internas o inferiores están interconectadas con las capas físicas, las capas intermedias con todo lo que es transferencia de datos y por último las capas superiores con transferencia de información. Al momento de diseñar el modelo OSI, el principal objetivo fue definir una implementación general para que cualquier sistema que esté acorde con ella se pueda comunicar entre sí.

Cada capa solamente debería depender de la capa inmediatamente inferior. A medida que se desarrollan nuevas tecnologías, una capa podría ser implementada de forma transparente sin afectar a los sistemas que usan el resto de las capas.

Debido a la existencia de muchas redes, una interconexión puede ser imposible. Las características funcionales están definidas en cada capa pero no las no funcionales. Entonces el deber de los desarrolladores es de crear sus propias facilidades y obviar algunas capas del modelo, logrando diseñar características para mejor el rendimiento de la apli-

cación.

Otro modelo de referencia que se usa es para herramientas CASE, la cual contiene cinco niveles de servicios: Servicios de repositorio de datos, servicios de integración de datos, servicios de gestión de tareas, servicios de mensajes y servicios de interfaz de usuario.

Este modelo de referencias se puede incluir en cualquier entorno CASE, elaborado cuestiones que creemos necesarias para el diseño de un sistema.

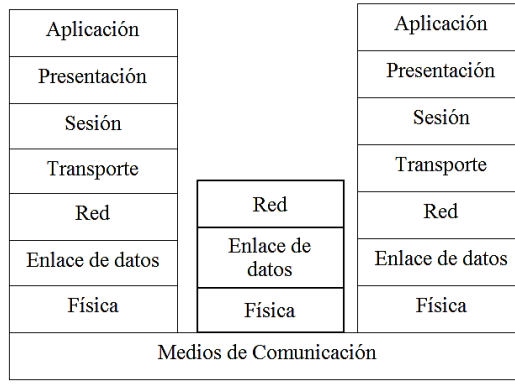



***Figura 10.*** Modelo OSI.

Esta arquitectura de referencia nos ayuda a clasificar y comparar entornos, utilizado especialmente como una herramienta CASE.

## 1.2 Arquitecturas de sistemas distribuidos.

Actualmente los grandes sistemas son distribuidos, ya que toda la información procesada es distribuida en varias computadoras, en vez de tenerla en una sola máquina como lo hacen los sistemas centralizados. Entre las ventajas que proporcionan los sistemas distribuidos tenemos:

1. Compartir recursos hardware y software a través de la red.
2. Son sistemas abiertos que se diseñan sobre un protocolo estándar.
3. Existe concurrencia, ya que varios procesos se ejecutan a la vez en diferentes computadoras.
4. Existe escalabilidad, ya que si es necesario se puede añadir nuevos recursos al sistema.

5. Posee mejor tolerancia a defectos de funcionamiento del hardware o software.

Las desventajas de los sistemas distribuidos en comparación con los sistemas centralizados son:

1.Los sistemas distribuidos son más complejos que los sistemas centralizados.

- 2Puede accederse desde varias máquinas, disminuyendo la seguridad de los datos ante ataques de denegación de servicios.
- Los problemas de una máquina pueden propagarse a otras.
- Los tiempos de respuesta a las peticiones de los usuarios varían drásticamente.

Se debe diseñar el sistema de tal manera que el software y el hardware proporcionen características favorables a los sistemas distribuidos y a su vez minimicen los problemas. Las arquitecturas de sistemas distribuidos más utilizadas son:

- Arquitectura cliente-servidor: Es un conjunto de servicios usados por los clientes, ambos elementos se tratan de forma diferente en el sistema.
- Arquitectura de objetos distribuidos: Son conjuntos de objetos que interactúan entre sí, no hay distinción entre servidor y cliente.

Todas estas arquitecturas son muy usadas en las empresas, pero su distribución se lo hace dentro de una única organización, existen otras arquitecturas como la pee-to-peer y las orientadas a servicios. Un sistema distribuido puede desarrollarse utilizando muchos lenguajes de programación y ejecutarse en distintos procesadores. Entonces, se requiere que el software este en distintas computadoras, asegurando la comunicación entre estas para poder realizar el respectivo intercambio de datos. Este software se lo conoce como middleware, situándose en medio de los componentes distribuidos del sistema.

### 1.2.1. Arquitecturas multiprocesador.

Un sistema multiprocesador se lo considera como el más simple de los sistemas distribuidos ya que el sistema se forma por varios procesos que pueden ejecutarse en diferentes procesadores. El modelo multiprocesador se lo usa en sistemas de tiempo real, también recoge información, de esa información toma decisiones y envía respuestas a los

actuadores, los cuales se dedican a modificar el entorno del sistema. Todas estas actividades se pueden ejecutar en un solo procesador pero estando bajo control de un planificador.

Usar varios procesadores notoriamente mejora el rendimiento en el sistema, se puede distribuir los procesos entre los diferentes procesadores de forma predeterminada o mediante el control de un despachador.

Un ejemplo de un sistema multiprocesador se encuentra en la figura 11 la cual nos indica sobre un sistema para el control del tráfico, donde un conjunto de sensores distribuidos almacena información para procesar y luego enviarlo a una sala de control, finalmente los

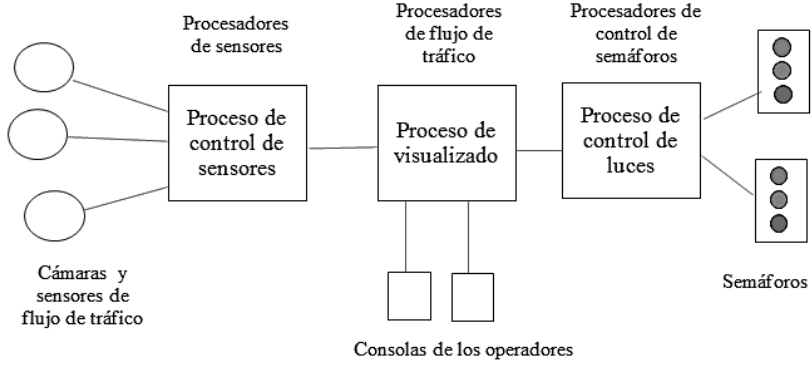



***Figura 11.*** Sistema multiprocesador para el control de tráfico.

operadores toman decisiones y envían las respectivas luces de los semáforos.

Si un software tiene muchos procesos no significa que siempre será un sistema distribuido. Para tener una distribución se necesita de más de un procesador. En el capítulo 2 se explicará la aproximación de diseño para aplicaciones de tiempo real.

### 1.2.2. Arquitecturas cliente-servidor.

En los temas anteriores ya se ha tratado brevemente sobre las arquitecturas cliente-servidor. Para recordar, estas son un conjunto de servicios que serán utilizados por un con conjunto de usuarios que los requieran.

Los clientes deben saber que servidores están libres, pero un cliente no sabe que otro cliente está queriendo acceder a los mismos servi-

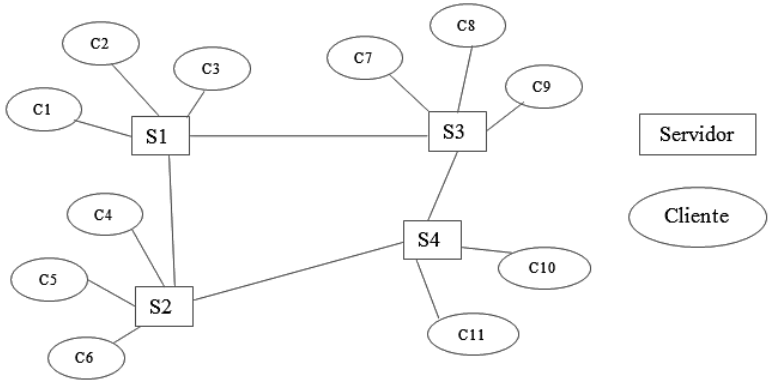



***Figura 12.*** Sistema cliente – servidor.

cios. Cliente y servidor son dos procesos distintos, en la figura 12 se logra evidenciar esto, mediante un modelo lógico.

Una arquitectura cliente - servidor representa la estructura lógica de la aplicación a desarrollar. Esto se detalla en la figura 13, donde observamos un sistema estructurado en tres capas. La capa de presentación permite la interacción y visualización de la información al usuario, la capa de procesamiento representa la lógica de la aplicación

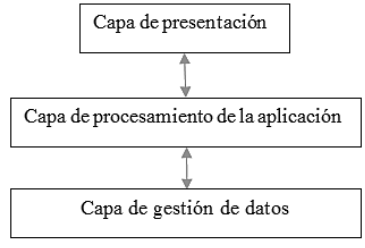



***Figura 13.*** Sistema de tres capas.

y la capa de gestión son todas las operaciones que se pueden realizar en la base de datos.

El modelo cliente - servidor más simple es el de dos capas, en donde una aplicación se establece como un servidor con muchos clientes.

Estas arquitecturas pueden ser de dos tipos, tal como se observa en la figura 14:

- Modelo de cliente ligero: El procesamiento de la aplicación como la gestión de los datos lo maneja el servidor, el cliente solo es la presentación del software.
- Modelo de cliente pesado: El servidor solo se dedica a gestionar

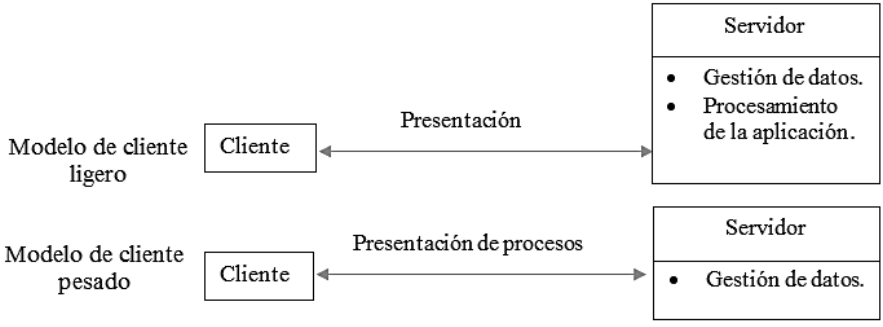



***Figura 14.*** Sistema de dos capas – Cliente ligero y pesado.

los datos, el cliente realiza la parte lógica de la aplicación y las interacciones con otros usuarios.

Un inconveniente del modelo de dos capas con clientes ligeros es la gran carga de procesos en el servidor y en la red, ya que el servidor realiza gran parte del trabajo, ocasionando la disminución del tiempo de respuesta para los clientes.

En cambio el modelo de dos capas con clientes pesados distribuye este trabajo, tanto de la parte lógica de la aplicación como la presentación al cliente. Un ejemplo de este modelo se observa en la figura 15, los sistemas bancarios ATM, donde el mainframe o servidor procesa la cuenta del cliente en la base de datos y cada ATM o cliente no se conecta directamente con esta, sino con el monitor de teleproceso, el cual se define como un sistema middleware que selecciona y ordena las comunicaciones de los clientes para realizar sus respectivas transacciones. El uso de transacciones serializadas permite que la aplicación se recu-

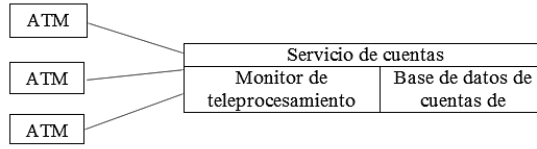



***Figura 15.*** Sistema bancario ATM.

pere de cualquier fallo sin dañar los datos del sistema. El inconveniente de este modelo radica en la complejidad de su gestión.

Con la aparición de lo móviles, se ha desarrollado aplicaciones intermedias entres los modelos de cliente ligero y pesado. Estas aplicaciones puedan descargase en el cliente como un código digital lo cual ayudada a disminuir la carga en el servidor.

Un modelo de dos capas con clientes ligeros tiene un problema en la escalabilidad o el rendimiento del sistema, ya que las tres capas lógicas deben unirse con dos computadoras para el cliente y el servidor, en el caso de un modelo de cliente pesado el problema estaría con la gestión del sistema. Como solución se podría usar una arquitectura cliente servidor de tres capas, como se la muestra en la figura 16, todas

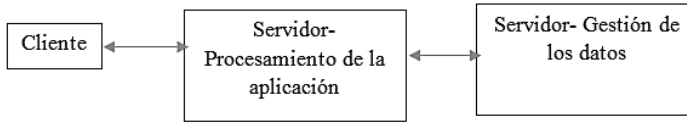



***Figura 16.*** Arquitectura cliente – servidor de tres capas.

estas capas son procesos lógicamente separados que se ejecutan en procesadores diferentes.

Para entender mejor un modelo cliente – servidor de tres capas, observamos en la figura 17 un sistema bancario por internet, su base de datos le proporciona al sistema un servicio de gestión de datos; un servidor web proporciona servicios de aplicación tales como transferir su dinero, generar estados de cuenta para sus clientes y pagar factura; el cliente es la computadora del usuario conectada a internet. Entonces se puede decir que este sistema es escalable debido a que se puede añadir nuevos servidores web siempre y cuando los clientes crezcan. Usar una arquitectura de tres capas permite al sistema optimizar toda transferencia que se dé entre el servidor y la base de dato; para recu-

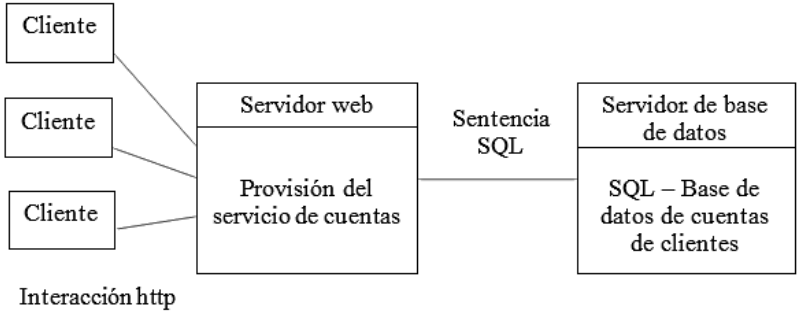



***Figura 17.*** Modelo cliente servidor de tres capas – Sistema bancario en internet.

perar la información se utiliza un middleware que sea compatible y soporte consultas SQL.

Cuando una aplicación necesite acceder o usar datos de algunas bases de datos, se puede usar un sistema multicapa, para esto se coloca un servidor de integración entre la aplicación y la base de datos, entonces el servidor de integración recoge los datos necesitados y los presentar al usuario como si se hubieran obtenido de una sola base de datos.

### 1.2.3. Arquitecturas de objetos distribuidos.

Los clientes y los servidores son diferentes cuando se trata de un cliente-servidor de un sistema distribuido. Los clientes reciben servicios de los servidores, no de los clientes y los servidores pueden ser clientes al recibir servicios de otro servidor. Cada cliente conoce los servicios que ofrecen los servidores y como contactarlos.

Los sistemas distribuidos tratan de eliminar la diferencia que existe entre cliente y servidor, diseñando algo que se conoce como arquitectura de objetos distribuidos, en la figura 18 se observa esta arquitectura, la cual está formada por objetos que realizan llamadas a estos servicios sin necesidad de distinción lógica entre los clientes y el servidor. Estos objetos están distribuidos en varias computadoras conectadas red, comunicándose a través de un middleware el cual brinda una interfaz transparente a todos los objetos existentes.

Las ventajas de este modelo son las siguientes:

- Permite al diseñador del sistema aplazar decisiones sobre dónde y cómo brindar estos servicios.

- Permite añadir nuevos recursos si es necesario.
- Permite añadir nuevos objetos a medida que la carga del sistema se incrementa sin afectar al resto de los objetos del sistema.

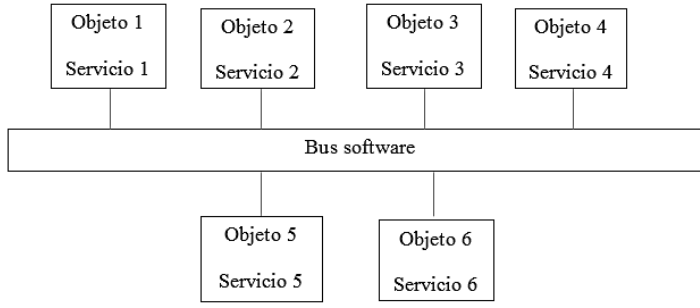



***Figura 18.*** Arquitectura de objetos distribuidos.

- Se puede reconfigurar el sistema dinámica usando la migración de objetos en red.

La arquitectura de objetos distribuidos nos ayuda a estructurar y organizar el sistema. Se debe proporcionar las funcionalidades del software en términos de servicios y sus combinaciones, para luego identificar como entregar estos servicios con la ayuda de varios objetos distribuidos.

De manera opcional se puede implementar sistemas cliente - servidor usando aproximaciones de objetos distribuidos, donde el modelo lógico del sistema vendría a ser un modelo cliente -servidor pero los clientes y servidores se deben implementar como objetos distribuidos, esto se hace con el fin de realizar cambios en el sistema de una manera más sencilla, por ejemplo se puede cambiar un sistema que tenga el modelo de dos capas a un sistema que tenga multicapas y para eso el servidor y el cliente no pueden ser un objeto distribuido único, al contrario debe estar compuesto por varios objetos para proporcionar servicios especializados.

Usar una arquitectura de objetos distribuidos es mejor que usar una aplicación cliente-servidor por las siguientes razones.

1. El modelo lógico del sistema no tiene provisión de servicios.
2. Se puede añadir bases de datos al sistema, ya que estas pueden estar en otro objeto distribuido.
3. Se puede añadir nuevas relaciones cada vez que se añade un nuevo

objeto.
La implementación de una arquitectura de objetos distribuidos, es más compleja que la de una arquitectura cliente - servidor.

#### 1.2.3.1. CORBA.

En la sección anterior se aprendió que para implementar la arquitectura de objetos distribuidos se necesita un middleware (software que permite intermediarios de peticiones entre objetos) para poder tener comunicación entre objetos. Antes, cada objeto en el sistema se lo implementaba usando cualquier tipo de lenguaje de programación y se podían ejecutar en diferentes sistemas operativos, por lo tanto el middleware nos facilita este trabajo.
Se utilizan dos niveles de middleware para la arquitectura de objetos distribuidos:

1. Nivel de comunicación lógica: El middleware nos otorga funcionalidades para que los objetos puedan intercambiar datos entre diferentes computadoras. Existen estándares que nos permiten tener una comunicación lógica entre objetos de distintas plataformas, entre ellas CORBA y COM.
2. Nivel de componentes: El middleware nos otorga una base para comenzar a desarrollar componentes que sean compatibles entre sí. Existen estándares como CORBA o Active X.

En esta nueva sección se hablará sobre el estándar CORBA y como soporta middleware para las comunicaciones lógicas de objetos. Estos estándares fueron creados por OMG (Object Mangament Group), los cuales están a cargo de todo lo relacionado al desarrollo orientado a objetos.
En la figura 1.19 se muestra como la visión de OMG se ha adapto al diagrama de Siegel de la arquitectura de administración de objetos, donde propone que un sistema distribuido este formado por los siguientes componentes:

- Los objetos de aplicación propios.
- Los objetos definidos por OMG.
- Los servicios fundamentales CORBA.
- Facilidades CORBA horizontales.

Existen cuatro elementos principales que incluyen los estándares

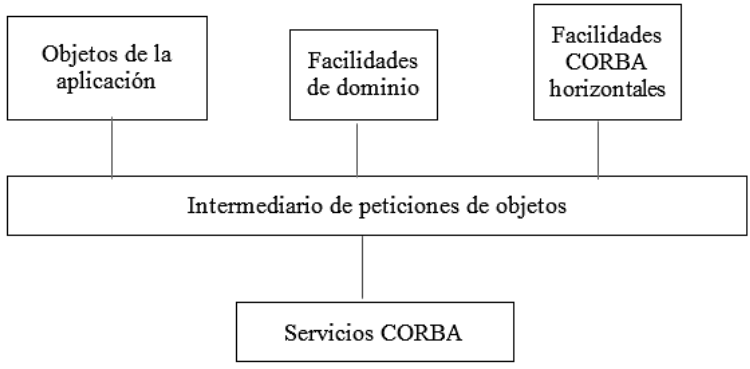



***Figura 19.*** Estructura de una aplicación distribuida basada en CORBA.

CORBA:

- Modelo de objetos CORBA.
  - Un intermediario de peticiones de objetos.
  - Conjunto de servicios de objetos.
  - Conjunto de componentes verticales u horizontales.

El modelo de objetos CORBA considera que un objeto es una encapsulación de atributos y servicios, sin embargo, cada objeto debe tener por separado una interfaz que lo defina. Si un objeto quiere tener los servicios de otro debe hacerlo usando la interfaz IDL. CORBA tiene un identificador llamado IOR, cuya función de es llamar a un objeto cuando este necesite los servicios de otro.

El intermediario de peticiones de objeto conoce que objeto está pidiendo usar los servicios y sus interfaces. Todos los objetos se comunican entre sí a través del ORB, al momento de comunicarse estos no necesitan saber cuál es su localización e implementación. En la figura 20 se observa dos objetos que se comunican usando ORB, donde el objeto 1 hace una llamada al stub IDL que define el servicio solicitado. El otro objeto que proporciona el servicio, tiene asociado un skeleton IDL, este se encarga de traducir las peticiones de las llamadas al lenguaje de implementación que esté utilizando. Cuando se ejecute el método, skeleton IDL se encarga de traducir estos resultados y los envía a IDL para que este pueda acceder al objeto que inicio la llamada. Si un objeto usa servicios de otra parte y también proporciona servicios a otros, necesita usar un skeleton IDL y un stub IDL.En un sistema distribuido

cada computadora tendrá su propio intermediario de peticiones, el cual manejará todas las invocaciones locales de objetos. Sin embargo, una

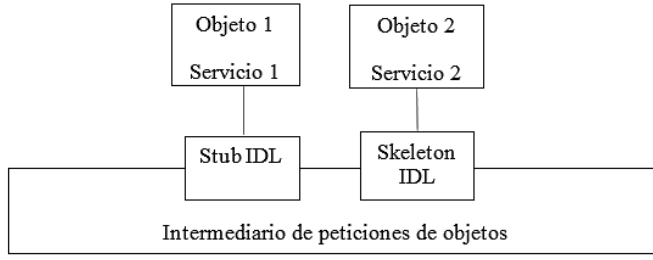



***Figura 20.*** Comunicación de objetos a través de ORB.

petición de un servicio proporcionado por un objeto remoto requiere comunicaciones inter-ORB.

CORBA se inició desde los años 80, sus primeras versiones solo daban soporte a los objetos distribuidos. Pero al pasar de los años se implementó un mecanismo de comunicaciones y se mejoró los estándares para que sirvan de soporte a las aplicaciones orientadas a objetos distribuidos.

Los estándares CORBA definen un conjunto de 15 servicios de objetos, de los cuales analizaremos tres:

1. Servicio de nombres y categorías: Permite que los objetos encuentren a otros dentro de la red, debido a que este servicio es un directorio donde muchos objetos están registrados realizando una búsqueda parecida a una agenda telefónica o a un DNS.
2. Servicio de notificación: Permite tener una notificación siempre y cuando ocurra algún evento que haya ocasionado un objeto, por ejemplo una cola de impresión donde están involucrados varios objetos impresora.
3. Servicio de transacciones: Ofrece transacciones atómicas y operaciones rollback, en el caso de que ocurra algún fallo.

### 1.2.4. Computación distribuida interorganizacional.

La seguridad que ofrecen ha sido el factor principal para que las empresas implementen la computación distribuida. Una empresa posee muchos servidores, los cuales se dedican a distribuir la carga de trabajo de las computadoras, algunos de estos servidores cuentan con están-

dares ya que se encuentran en la misma organización. Aunque para los sistemas web, tanto los servidores como los clientes están fuera de la empresa y su funcionalidad estaría limitada. Los modelos actuales permiten una computación distribuida interorganizacional, una de ellas es la arquitectura peer-to-peer y las orientadas a servicios.

#### 1.2.4.1. Arquitectura peer-to-peer (p2p).

Son sistemas descentralizados, que realizan cálculos en cualquier nodo de la red y no existe diferencia entre clientes y servidores. Esta arquitectura fue diseñada para sacar provecho de toda la potencia de una red de computadoras. Al principio fueron diseñados para entornos personales, como ejemplo el protocolo GNutella que se encarga de compartir archivos a otros usuarios.

El diseño de esta arquitectura ha incrementado su implementación en empresas, por ejemplo las empresas Intel y Boeing se decidieron a usar sistemas p2p. Si usan aplicaciones que soportan un trabajo distribuido este tipo de arquitectura es la más ideal.

P2P tienen una arquitectura lógica o arquitectura del sistema y una arquitectura de aplicación o componentes de la aplicación. En este tema estudiaremos los dos tipos de arquitecturas lógicas, las descentralizadas y las semicentralizadas.

Al comienzo, en los sistemas peer-to-peer cada nodo podía saber qué es lo que estaba haciendo el otro, conectarse entre sí e intercambiar datos, pero eso es en teoría, en la práctica es imposible debido a que los nodos están dentro de localidades con otros nodos los cuales sirven como puente para ir a otras localidades de nodos. En una arquitectura descentralizada, sus nodos no solo son elementos funcionales ya que también sirven para ser interruptores de comunicaciones y dar

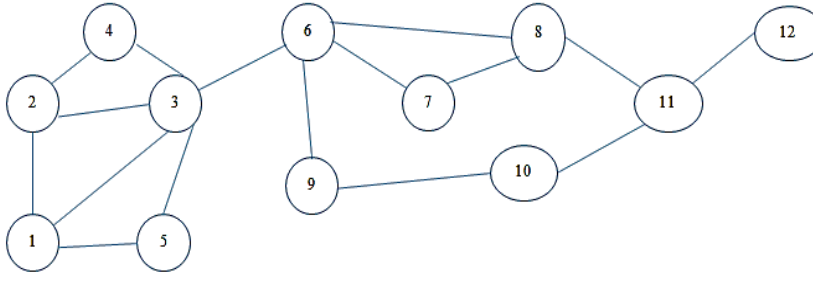



***Figura 21.*** Arquitectura peer-to-peer descentralizada.

señales de control entre sí. Supongamos que la figura 21 representa un sistema de gestión de documentos descentralizado.

Una arquitectura descentraliza tiene varias ventajas, es tolerante a defectos y a nodos desconectados de la red. Una desventaja seria la sobrecarga en el sistema ya que las búsquedas pueden ser procesadas por muchos nodos diferentes.

Una arquitectura p2p alternativa es la semicentralizada, dentro de la red uno o más nodos actúan como servidores para facilitar la comunicación entre ellos. La función principal del servidor es tener contacto entre iguales en la red y coordinar los resultados. La figura 22 representa un sistema de mensajería instantánea, donde cada nodo de la red se comunica con el servidor con el objetivo de encontrar nodos

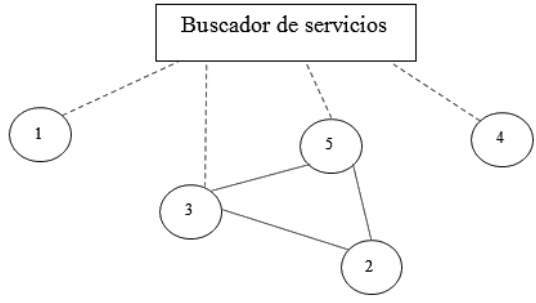



***Figura 22.*** Arquitectura peer-to-peer semicentralizada

que estén disponibles, si se encuentran se establece una comunicación directa y la comunicación con el servidor ya no es necesaria.

**1.2.4.2. Arquitectura de sistemas orientados a servicios.**

El internet hizo posible que las computadoras clientes accedieran a servidores alojados fuera de una empresa. Solo se necesitaba aplicaciones hechas con HTML para que cualquier cliente pueda acceder desde sus casas. Pero este acceso al sistema no era práctico, ya que solo se requería de un navegador web o utilizar otros programas para acceder a la información.

Entonces para poder solucionar el problema, surgió la idea de tener un servicio web, permitiendo a las empresas volver accesible su información a otros programas, definiendo y publicando una interfaz de servicio web. Es decir, se define a un servicio web como aquella representación que puede ser utiliza como recurso computacional.

Algunos proveedores de servicios pueden ofrecer servicios especializados y darles esta promoción a los usuarios de diferentes empresas. Una aplicación puede funcionar enlazando servicios desde varios proveedores, utilizando un estándar de lenguaje de programación.

Las diferencias que existen entre un modelo de servicio con un modelo de objetos distribuidos son las siguientes:

- Un servicio lo puedo ofrecer cualquier proveedor ya sea dentro o fuera de la empresa. Estas empresas puede realizar aplicaciones en las cuales integren los servicios de muchos proveedores.
- Un proveedor ofrece su servicio para que otro usuario lo pueda usar, no es necesario establecer una negociación entre los proveedores de servicios con el cliente.
- Las aplicaciones pueden retrasar el enlace de los servicios hasta que éstas sean desplegadas o hasta que estén en ejecución.
- Si una aplicación está en ejecución ocasionaría un retraso en el enlace de estos servicios.
- Se puede crear nuevos servicios, ya que un proveedor puede enlazar servicios ya existentes en la empresa y crear algo innovador.
- Si se implementan el manejo de excepciones en las aplicaciones como un servicio externo, esto podría disminuir el tamaño de la aplicación.

Los tres estándares fundamentales que permiten la comunicación entre servicios web son:

1. SOAP.
2. WSDL.
3. UDDI.

Todos estos estándares se basan en XML, u un lenguaje comprensible para el humano y la máquina. Sin embargo, no se necesita conocer a fondo estos estándares para tener una noción de lo que es un servicio web. La arquitectura que usan los servicios web esta acoplada de una manera muy débil debido a que el enlace de los servicios pueden variar cuando el sistema está en ejecución. La arquitectura orientada a servicios no es posible de implementar con los servicios web actuales, debido a que los enlaces de servicios son muy estáticos. No cabe ninguna duda de que la arquitectura orientada a servicios es una muy buena manera de implementar sistemas distribuidos dentro de las empresas.

**1.3 Arquitecturas de aplicaciones.**

En esta sección se presenta modelos genéricos para muchos tipos de aplicaciones. Se va a estudiar la organización básica que poseen estas aplicaciones y cuando es apropiado utilizarlos. Además se va a descomponer una arquitectura de alto nivel para ver los subsistemas que conforman estas aplicaciones.

Existen muchas arquitecturas de aplicaciones genéricas que como programador se pueden utilizar, las que estudiaremos son las siguientes:

1. Como punto de inicio al proceso de diseño arquitectónico: Si no se tiene experiencia se puede trabajar sobre diseños de arquitecturas genéricas.
2. Como una lista de comprobación: Sirve para colaborar con la arquitectura de aplicación genérica, y verificar si falta algún componente de diseño a su sistema.
3. Como una forma para organizar su trabajo: Al permitir el trabajo en paralelo se puede asignar trabajo a miembros del grupo para implementar diferentes subsistemas dentro de la arquitectura.
4. Como una forma de evaluar los componentes para su reutilización: Comparar los componentes con las estructuras genéricas para ver si es probable la reutilización en la aplicación a desarrollar.
5. Para usarlo como vocabulario entre aplicaciones: Comparar aplicaciones del mismo tipo usando las estructuras genéricas.

Existen cuatro aplicaciones con extensos diseños arquitectónicos:

1. Procesamiento de datos.
2. Procesamiento de transacciones.
3. Procesamiento de eventos.
4. Procesamiento de lenguajes.

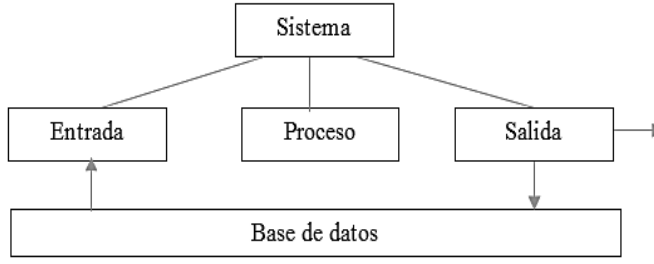
Todas estas aplicaciones se usan en la actualidad, por ejemplo en sistemas de negocio se usa los de procesamiento de transacciones y de datos, en cuanto al procesamiento de eventos, se lo utiliza en los sistemas de tiempo real.

**1.3.1. Sistemas de procesamiento de datos.**

Estos sistemas son de mucha ayuda para los negocios, debido a que gestionan los procesos de pago de salarios, cálculos de facturas, entre

otros. Estos sistemas son considerados como procesamientos por lotes, ya que cada dato es incluido y extraído de una base de datos o un fichero en forma de lote.

El procesamiento por lotes se divide en tres componentes, tal como se observa en la figura 23, la entrada reúne datos de una o más fuentes, el proceso de datos se encarga de realizar los cálculos usando las entradas obtenidas y la salida genera un resultado para guardarlo en una base de dato. Un ejemplo de este sistema es el de facturas electrónicas, ya que toma los datos de un cliente y de los productos de la base de datos (entrada), luego calcula los valores correspondientes al




valor a pagar (proceso) y por último procede a imprimir el comprobante (salida).

Ahora se detallará ejemplos de cómo trabajan estos componentes:

- Componente de entrada: Se encarga de leer datos ya sea de un fichero o de una base de datos.
- Componente de proceso: Realiza algún cálculo de acuerdo a las entradas especificadas.
- Componente de salida: Son los resultados que se procederán a imprimir o guardar en una base de datos.

Usar un diagrama de flujo es muy útil para describir el funcionamiento de los sistemas de procesamiento de datos. Una de las ventajas de utilizar diagramas de flujo para representar estos sistemas es que se puede visualizar todo el procesamiento de datos desde que inicia la entrada hasta que finaliza (salida).

Su modelo arquitectónico es muy fácil de implementar, la parte compleja de este sistema radica en los datos que se van a procesar, entonces cuando se va a diseñar la arquitectura del sistema se debe pensar en la arquitectura de los datos así como se piensa en la arqui-

tectura de la aplicación.

### 1.3.2. Sistemas de procesamiento de transacciones.

Estos sistemas fueron diseñados para procesar peticiones de los usuarios con el objetivo de obtener o actualizar información de una base de datos. Una transacción es una secuencia de operaciones que se la toma como una unidad atómica, todas estas operaciones deben ser confirmadas antes de realizar el guardado en la base de datos. Como un ejemplo de transacción se puede mencionar a un cliente solicitar el retiro de su dinero utilizando un cajero automático, para esto se necesita saber la cuenta del cliente, ver cuánto saldo tiene, modificar su saldo y dar el dinero respectivo, hasta que no se cumpla todo esto la base de datos no se debe modificar.

En la figura 24 se observa como es la estructura de aplicaciones del procesamiento de transacciones, donde un usuario realiza una petición utilizando el procesamiento e/s, esta petición es procesada por

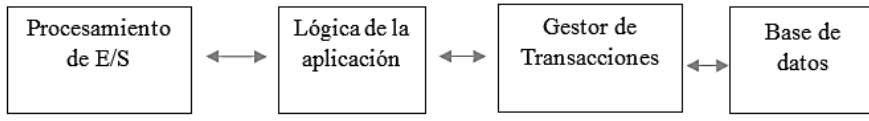



***Figura 24.*** Estructura de aplicaciones del procesamiento de transacciones.

el componente de la lógica de la aplicación, para luego enviar esta transacción a su gestor, el cual funciona utilizando una base de datos.

La estructura de entrada-proceso y salida se lo ve en muchas aplicaciones de transacciones y de procesamientos de datos, muchos de estos sistemas son interactivos y funcionan en un sistema de procesamiento por lotes, por ejemplo en los bancos cuando los clientes durante el día realiza transacciones y en la noche estas transacciones se ejecutan por lotes en conjunto con la base de datos. Este proceso se lo observa en la figura 25.

Otro ejemplo de este sistema de transacción es un sistema de cuentas bancarias, en donde los usuarios pueden extraer su dinero de un cajero automático, el cual está formado por dos ATM y una base de datos. En la figura 26 se observa dicho proceso, en donde se usa un monitor de teleprocesamiento para manejar entradas y serializar tran-

sacciones, las cuales le sirve a la base de datos como consultas. Entonces se puede decir que el procesamiento de consultas se da en la base de datos. Después estos resultados se los vuelve a enviar al monitor de teleprocesamiento y este se va a encargar de registrar los terminales

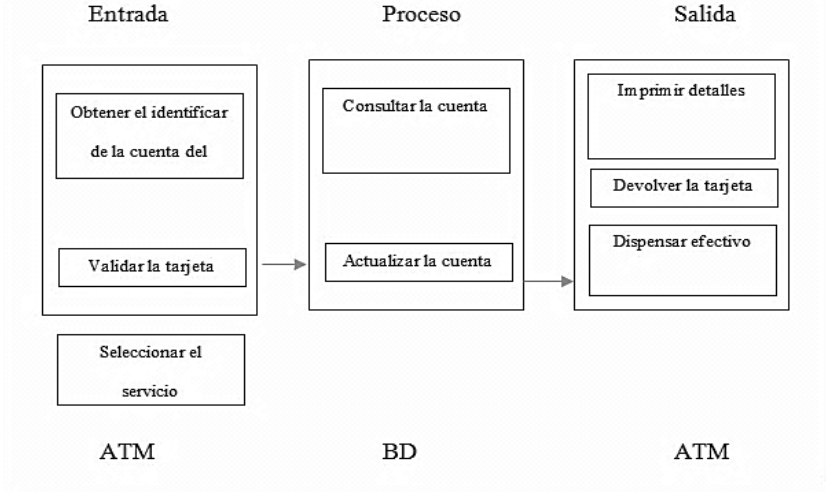



***Figura 25.*** Sistema de transacciones bancarias usando un ATM.

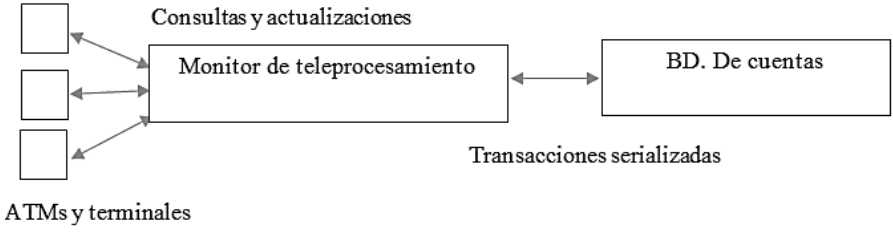



***Figura 26.*** Middleware para gestión de transacciones.

que han solicitado estas peticiones. Al final el sistema se encargará de organizar estos datos y entregar el resultado de la transacción.

**1.3.2.1. Sistemas de información y de gestión de recursos.**

Si existe una interacción entre un sistema con la base de datos compartida entonces este sistema puede ser uno basado en transacciones. El objetivo de un sistema de información es dar acceso a una gran cantidad de información, como un sistema de biblioteca o los registros de

trabajadores. La figura 27 representa un modelo general de un sistema

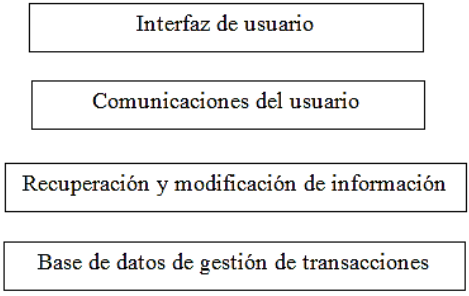



***Figura 27.*** Sistema de información.

de información, donde cada capa superior soporta la interfaz de usuario y la capa inferior la base de datos del sistema.

En la figura 28 se muestra la arquitectura por capas del sistema de bibliotecas LYBYS. Si se analiza la capa de comunicación usted puede observar tres componentes importantes:

2. Identificación: Permite la autenticación de usuarios.
3. Consultas y formularios: Permite realizar búsquedas que el usuario necesite.
4. Gestión de impresión: Controla las impresiones de los libros ya que

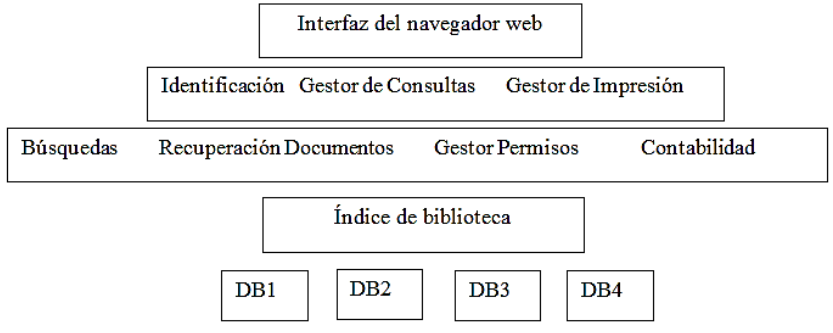



***Figura 28.*** Sistema bibliotecario LYBYS.

cada libro tiene un derecho de autor, la impresión de un libro puede ser no posible, o bien puede ser impreso solo una vez.
En la capa de recuperación de información se incluye los siguientes componentes:
1. Búsquedas: Permite realizar búsquedas de documentos que el usuario desea consultar, todos los libros están registrados en la base de datos.
2. Recuperación de documentos: El usuario podrá recuperar un documento que haya leído anteriormente.
3. Gestor de permisos: Gestionar los libros digitales y derechos de autor, un usuario no puede abusar de las búsquedas del mismo libro.
4. Registro de cuentas: Registrar las solicitudes de libros que los usuarios sugieren.
Una de aplicaciones muy usadas son los llamados sistemas de asignación de recursos, estos tipos de sistemas se asemejan a un sistema de

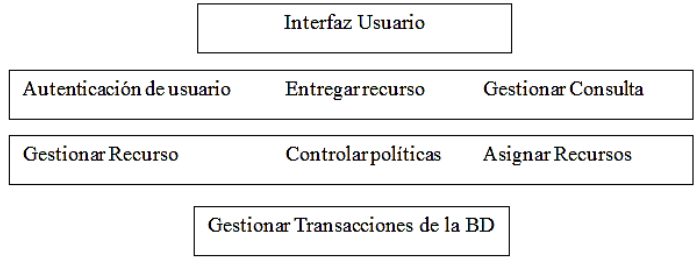



***Figura 29.*** Capas de un sistema de asignación de recursos.

información, en la figura 29 se observa las capas que conforman un sistema de asignación de recursos.
Esta arquitectura por capas puede considerarse de varias formas. Por ejemplo en un software de sistemas de información cada capa puede ser un componente de gran escala que se ejecute en un servidor distinto. Cada capa tiene interfaces externas y la comunicación que se da través de estas interfaces. En el caso de que el sistema se ejecute en una sola computadora, las capas de en medio se las puede implementar de tal forma que se haga un solo programa, permitiendo la comunicación directa con la base de datos usando sus Apis.

La interfaz gráfica en un sistema de información y de gestión de recursos se la implementa usando un navegador web. Como se observa

en la figura 30, un servidor web se encarga de las comunicaciones que el usuario hace, y un servidor de aplicaciones se encarga de integrar la lógica de la aplicación. Un servidor de base de datos se encarga de trasladar la información hasta la base de datos. Al usar varios servidores

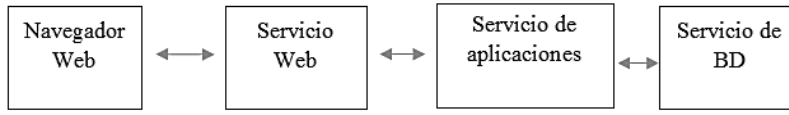



***Figura 30.*** Sistema multicapa de procesamiento de transacciones por Internet.

aumentará el rendimiento del sistema, ejecutando transacciones con mayor rapidez.

### 1.3.3. Sistemas de procesamiento de eventos.

La característica más destacable del procesamiento de eventos es que son impredecibles. Estos sistemas deben ser capaces de controlarlos cuando sucedan. Algunos sistemas como estos son los procesadores de texto, juegos y de presentación, en donde el sistema detecta e interpreta los eventos.

Los sistemas en tiempo real, son otro claro ejemplo de este tipo de procesos, pero este se diferencia porque los eventos están asociados con actuadores del sistema o servidores, como se debe tener una respuesta para este tipo de eventos existen conjuntos de procesos cooperativos.

Los sistemas de edición nos permiten editar documentos tales como imágenes, texto, entre otros. Algunos de estos tienen funciones específicas. Estos sistemas de edición tienen características que las diferencian de los demás sistemas:

1. Este tipo de sistema es usado por un único usuario a la vez, evitando lidiar con los múltiples accesos y la concurrencia en la aplicación.
2. Deben permitir la recuperación de la información almacenada en la memoria volátil al ocurrir un fallo en el sistema.
3. Debe existir facilidad de guardado y recuperación automática de información.

La arquitectura genérica se la define como un grupo de objetos que interactúan entre sí, y estos objetos pueden ser activos o pasivos, los cuales operan concurrentemente y actualizan la estructura de datos. Estos componentes se las pueden visualizar en la Figura 31:

1. Pantalla. Estos eventos detecta las coordenadas de la pantalla y son

enviados a un objeto que se encarga del procesamiento de los eventos
2. Evento. Este ente se origina por un evento desde la pantalla, luego es enviado a un interpretador de comando el mismo que detecta acciones por el mouse, el teclado u otro objeto.
3. Orden. Este objeto procesa los comandos originados por el objeto accionado y hace una llamada al método idóneo que ejecute el proceso.
4. Editor de datos. Este comando sirve para actualizar y visualizar los datos ya modificados.
5. Datos auxiliares. Los editores también controlan otras funciones como son los estilos y preferencias del usuario, como ejemplo se tiene la revisión ortográfica.
6. Sistema de ficheros.

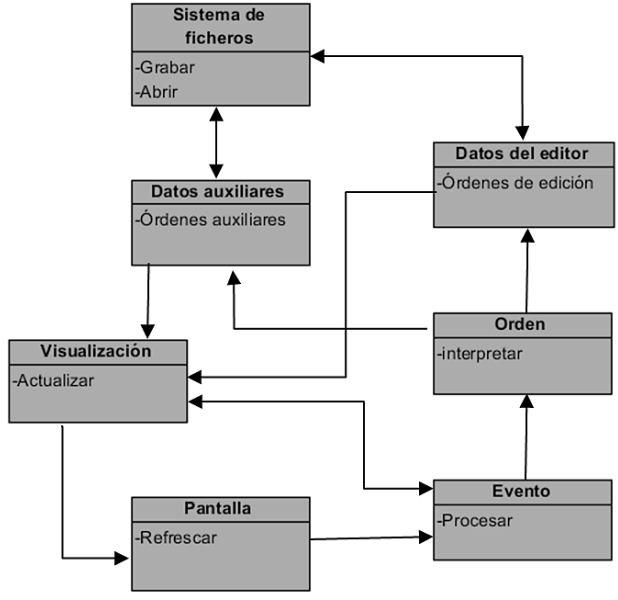



***Figura 31.*** Modelo arquitectónico de un sistema edición.

7. Visualización. Este se encarga de visualizar los objetos y presentar los últimos cambios realizados.

### 1.3.4. Sistemas de procesamiento de lenguajes.

Estos sistemas de procesamiento de lenguajes son como traductores, aceptan como entrada un tipo de lenguaje natural o artificial y pre-

senta como salida otro lenguaje, estos sistemas principalmente son usados en los compiladores, ya que estos aceptan un lenguaje de alto nivel en su entrada y lo traduce a un lenguaje ensamblador para que lo interprete la máquina.

A continuación en la figura 32 se puede observar una arquitectura de un sistema de procesamiento de lenguajes, donde las instrucciones detallarán que tiene que traducirse y bajo qué formato interno trabajará, después estas son interpretadas por algún otro componente, el

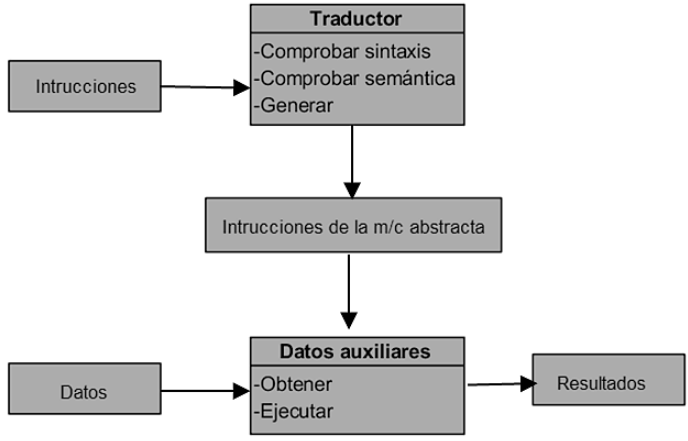



***Figura 32.*** Arquitectura de un sistema de procesamiento de lenguajes.

mismo que ejecuta según las instrucciones dadas y da como salida los datos ya interpretados.

Estos sistemas de procesamiento son usados principalmente en herramientas meta-CASE es decir, herramientas capaces de crear herramientas CASE, en las cuales se especifica la solución como una descripción de los datos, reglas y analizadas sintáctica y semánticamente. En la Figura 33 se puede visualizar la arquitectura genérica de los traductores:

1. Usa un analizador léxico: Posee entradas de símbolos de lenguaje y este pasa a convertirse en un formato interno.
2. Una tabla de símbolos: Contiene información sobre los objetos a traducir.
3. Un analizador sintáctico: Verifica la sintaxis del lenguaje.
4. Un árbol sintáctico: indica como es la estructura interna del progra-

ma.

5. Un analizador semántico: Comprueba si el texto de lenguaje de entrada está correctamente escrito, utilizando al árbol sintáctico y la tabla de símbolos.

6. Un generador de código: Recorre el árbol sintáctico y genera código de máquina abstracta.

Los componentes de un sistema de procesamiento de lenguaje se pueden ordenar en varios modelos arquitectónicos existentes, en Figura 33 se puede visualizar una arquitectura de flujo de datos con la tabla de símbolos como un repositorio en común.

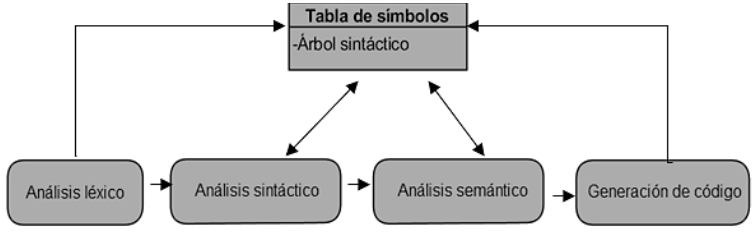



***Figura 33.*** Un modelo de flujo de datos de un compilador.

Este tipo de modelo es muy apto en entornos de procesamiento por lotes, en donde los programas son compilados automáticamente sin intervención del usuario. También los componentes genéricos se los pueden ordenar en un modelo basado en repositorios como se observa

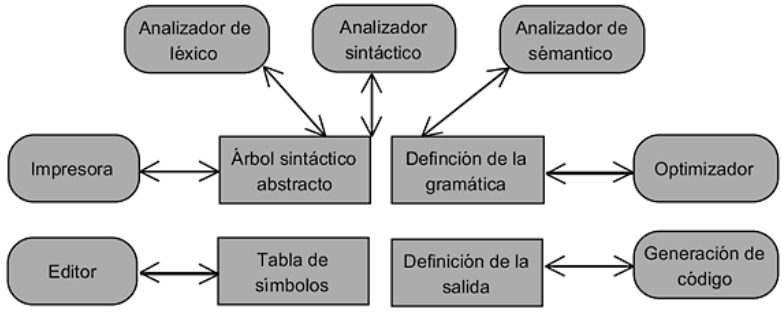



***Figura 34.*** El modelo de repositorio de un sistema de procesamiento de lenguajes.

en la figura 34, donde la tabla de símbolos y el árbol sintáctico funcionan como un repositorio en común.

**Ejercicios**

a) Describa a los sistemas distribuidos e indique por que son escalables.
b) Compare e indique las diferencias entre la arquitectura cliente-servidor ligero con el cliente-servidor pesado.
c) Descrtiba los sistemas de Informacion y gestion de recursos
d) Indique la estructura de aplicaciones del procesamiento de transacciones
e) Describa los componentes de la capa de recuperación de información en los sistemas de información
f) Indique que ventajas brinda un enlace dinámico
g) Mencione ventajas y desventajas de los modelos peer-to-peer, centralizadas y cliente-servidor.
h) Indique ¿por que puede provocarse cambios en la base de datos al momento de ingresar datos utilizando transacciones?

# Resumen

- Para elegir el diseño arquitectónico que se va a utilizar en el desarrollo de una aplicación, se debe tener en cuenta el tipo de sistema, estilos y distribución.
- Durante el proceso arquitectónico se pueden desarrollar modelos como el estructural, de control, de descomposición.
- El marco fundamental para estructurar un sistema es su arquitectura. El rendimiento, seguridad y disponibilidad del sistema, influyen para elegir una.
- Se puede compartir recursos con un sistema distribuido, también son escalables y tolerantes a fallos.
- El modelo cliente servidor es un conjunto de servicios que ofrecen los servidores para que los clientes lo usen.
- Para gestionar comunicaciones entre objetos se usa un software llamado middleware y ayuda a que los objetos se puedan agregar o eliminar del sistema.
- CORBA se lo define como un conjunto de estándares para que el middleware pueda soportar objetos distribuidos.
- La mayoría de aplicaciones pertenecen al grupo de las aplicaciones genéricas, estas aplicaciones genéricas se las consideren como sistemas para procesar datos o transacciones, eventos o de lenguajes.
- Tener un modelo genérico nos ayudará a comprender como funciona las aplicaciones y hacer comparaciones de aplicaciones que sean del mismo tipo.

# Modelado del sistema

En este capítulo 2 del Modelado del sistema, se estudian los temas que a continuación se detallan:

El punto 2.1 con el tema Diseño orientado a objetos tiene como objetivo principal producir un diseño que interconecta objetos de datos y operaciones de procesamiento para esos objetos, de forma que se modulariza la información y el procesamiento, en lugar de aislarlo. Los subtemas a estudiar en este punto se destacan las más importantes:

- Objetos y clases
- Un proceso de diseño orientado a objetos
- Evolución del diseño

En el punto 2.2 del Diseño de software de tiempo real tiene como objetivo principal Los subtemas más relevantes a tratar son:

- Diseño del sistema
- Sistemas operativos de tiempo real
- Sistemas de monitorización y control
- Sistemas de adquisición de datos

En el punto 2.3 del Diseño de interfaces de usuario tiene como objetivo principal introducir algunos aspectos de bosquejo de las interfaces de usuario que son significativas para los especialistas de software. Entre los temas a estudiar en el diseño de las interfaces son:

- Asuntos de diseño
- El proceso de diseño de la interfaz de usuario
- Análisis del usuario
- Prototipo de la interfaz de usuario
- Evaluación de la interfaz

## 2.1. Diseño Orientado a Objetos.

Un sistema orientado a objetos es aquel formado por objetos relacionados entre sí, que mantienen un estado local permitiendo realizar operaciones sobre el mismo (Figura 35). Su estado es representado como privado lo que implica que no se puede acceder a él desde afuera. Su proceso de diseño comprende el diseño de sus clases y las relaciones entre las mismas. Los objetos del sistema y sus interacciones son definidos por sus clases. Cuando este diseño es implementado en un programa ejecutable, los objetos se crean dinámicamente gracias a las definiciones de las clases.

El diseño orientado a objetos forma parte del desarrollo orientado a objetos en cualquier tipo de lenguaje de programación, mediante su proceso de desarrollo usa estrategias orientadas a objetos.

- El análisis orientado a objetos interpreta el dominio de la aplicación mediante el desarrollo de un modelo orientado a objetos. Los objetos establecidos reflejan las operaciones y entidades que están relacionadas con el problema.
- El diseño orientado a objetos desarrollo un modelo que se encarga de implementar los requerimientos identificados de un sistema con el fin de dar solución al problema a resolver. La relación entre objetos del problema con objetos de la solución puede existir en el diseño, pero necesariamente el diseñador debe incluir nuevos objetos para cambiar los objetos del problema y dar alternativas de solución.
- La programación orientada a objetos se trata de la implementación del diseño de un sistema en un lenguaje de programación orientado a objetos como es el caso de java. Un lenguaje de programación consta con los procesos necesarios para definir clases y un sistema para crear los objetos de dichas clases.

El cambio de una etapa a otra se lleva a cabo sin problema usando metamodelos compatibles entre ellas. Para pasar de una etapa a otra se debe mejorar la etapa anterior, detalles a las clases actuales y crear nuevas clases con el objetivo de proporcionar nuevas funcionalidades. Debido a que la información se esconde dentro de los objetos, puede atrasarse el diseño detallado de como representar los datos hasta que el sistema este implementado, la distribución de los objetos y la deci-

sión de si se implementan de forma secuencial o concurrente en algunos casos pueden presentar retrasos.

Esto nos quiere decir que los diseñadores desconocen los detalles de la implementación del sistema, pueden crear diseños que se adapten a los diversos entornos de ejecución. Un ejemplo de esto es el enfoque de arquitectura dirigida por el modelo (MDA), donde nos propone diseñar el software en dos niveles, uno que no dependa de la implementación y otro que sí dependa de ésta.

En el nivel independiente se diseña un modelo abstracto dirigido a un modelo dependiente de la plataforma con mayor detalle, a partir de cual se puede generar códigos. En la actualidad MDA es experimental todavía, ya que aún no se logra comprender el nivel de adopción

El mantenimiento de los sistemas orientados es fácil, porque un objeto no es dependiente de otro, es decir que se los puede considerar y modificar como entidades independientes. Por tal razón las modificaciones que se realicen en los objetos en su implementación o agregar algún servicio adicional al mismo no deben afectar a los demás objetos existentes en el sistema, debido a que frecuentemente los objetos están relacionados entre las entidades del mundo real (elementos de hardware) y objetos encargados del control del sistema.

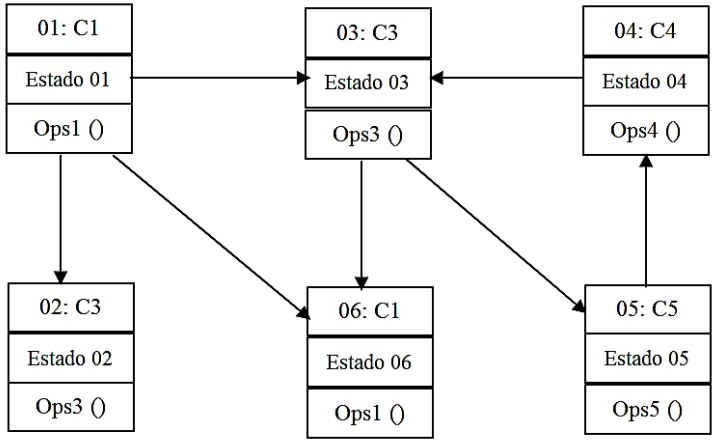



***Figura 35.*** Un Sistema compuesto de objetos que interactúan entre sí.

Los objetos de un sistema son considerados reutilizables por ser encapsulamientos independientes de los estados y sus operaciones. Por tal razón en el desarrollo de un sistema se pueden usar objetos de otro sistema o tomar objetos estándar, lo cual implica la reducción de los costos de diseño, programación, validación y los riesgos de desarrollo de software. Algunas veces para mejor beneficio del uso de la reutilización se pueden implementar colecciones de objeto.

Cuando un sistema de desarrollo está orientado en un diseño amplio que implica un extenso esfuerzo de análisis y diseño y no a un diseño de desarrollo incremental, se aplica los llamados métodos de desarrollo ágiles quienes reducen la actividad de un diseño orientado a objetos. Este modelo de desarrollo puede ser aplicado a sistema de negocio pequeño o mediano ya que para este tipo de negocios no es necesario un diseño pesado.

Para el desarrollo de sistemas críticos es importante asegurar que los equipos trabajen coordinadnos en todas partes del sistema. Por esta razón para explicar esta metodología de desarrollo se va usar como ejemplo un sistema donde es de mayor utilidad el enfoque del diseño orientado a objetos.

Esta percepción es reflejada en el Proceso Relacional Unificado (RUP) que orienta a grandes sistemas de software el desarrollo iterativo e incremental. RUP es un proceso iterativo expresa los requerimientos y el diseño de desarrollo orientado a objetos mediante casos de uso enfocando en el diseño de la arquitectura.

En la sección 2.1 se detalla un proceso que tiene algunos detalles en común con el RUP pero en su desarrollo no se aplica mucho los casos de uso. Los casos de uso determinan que el diseño este basado en los requerimientos del usuario y las interacciones con el sistema. Por ello es dificultoso representar los requerimientos que no están relacionados directamente con el usuario mediante casos de uso. Los casos de usos necesitan otras técnicas para descubrir los requerimientos que no son expresados directamente por los usuarios, requerimientos no funcionales.

### 2.1.1. Objetos y clases.

Actualmente las expresiones “objetos” y “orientados a objetos” se suelen aplicar a varios tipos de entidades, lenguajes de programación, métodos

de diseño. En las siguientes definiciones, se puede comprender que el objeto es un encapsulamiento de la información:

Se denomina objeto a una entidad que posee un estado y grupo definido de operaciones sobre ese estado, el cual es representado como el conjunto de atributos del objeto. Los servicios que ofrecen las operaciones de un objeto a otro objeto (clientes), son llamados cuando se necesita realizar algún tipo de cálculo.

Los objetos son creados a partir de la definición de una clase. Cuya definición es usada como plantilla para crear objetos, y en ella están incluidos los atributos y operaciones de están relacionados a objetos de dicha clase.

En UML (Lenguaje de Modelado Unificado) la clase es representado mediante un rectángulo, compuesto por tres secciones, en la parte superior se ubica en nombre de la clase, en la sección intermedia los atributos y en la parte inferior las operaciones asociadas con el objeto. En la figura 36 se observa el modelado de una clase de objetos de un empleado de una organización. En el diseño UML la operación hace referencia a una acción y un método a la implementación de una operación.

En la clase Empleado (Employee) se detallan los atributos de la clase con la información de los empleados, información como nombre, dirección, entre otros. Mediante los puntos suspensivos (…), indicamos que en esta clase existes más atributos. Las operaciones son Join (ingresa un empleado), Leave (elimina un empleado de la organización), Retire (convierte un empleado en pensionista), ChangeDetails (modifica información de un empleado).

La comunicación de los objetos se realiza llamando a los métodos (solicitud de servicio) de otros objetos e intercambiando la información requerida en caso de ser necesario para suministrar ese servicio. Son enviados como parámetros los resultados de la ejecución y las copias de información necesarias para dicha ejecución, ejemplos de estos estilos de comunicación:

```
//Llama a un método asociado con un objeto búfer
// que regresa el siguiente valor en el búfer
v = circula rBuffer.Get 0;
//Llama al método asociado con un objeto
// termostato que establece la temperatura que se mantendrá
thermostatsetTemo(20);
```

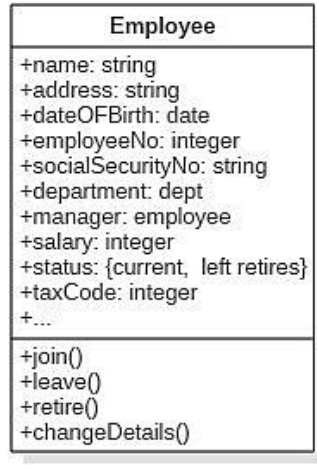



***Figura 36.*** Un objeto empleado.

Para comunicarse entre objetos en los sistemas basados en servicios, se intercambian directamente mensajes de texto; el objeto que recibe el mensaje es el encargado de examinar el mensaje, identificar datos y los servicios, para procesar el servicio solicitado. En un lenguaje como C, cuando hay objetos que se encuentran en el mismo programa el enlace a los métodos son implementados como enlaces a las funciones o procedimientos de dicho lenguaje.

La comunicación entre los objetos es síncrona en el momento que las solicitudes de los servicios son implementadas siguiendo esta ruta, la cual se trata de que el objeto emisor espera que se complete la petición del servicio. La comunicación se vuelve asíncrona en el momento que los objetos son implementados como procesos concurrentes, y cuando ejecutan el servicio solicitado el objeto emisor sigue realizando sus operaciones. Más adelante en este capítulo se detallará como se implementan los procesos concurrentes en los objetos.

Las clases se la puede organizar por medio de una jerárquica de herencias donde las clases generales deben tener relación con las especificas es decir las clases se pueden generalizar. En el diseño UML, la generalización se representa con una flecha que apunta a la clase padre, en los sistemas orientados a objetos se implementa mediante la herencia, donde la clase hija hereda operaciones de la clase padre. Se

observa un ejemplo de jerarquía de clases en la figura 37 (Clase Employee). Las clases hijas (clases inferiores) heredan los mismos atributos que las clases padre, atributos que pueden ser modificados desde ellas además que se le pueden agregar nuevos atributos a las clases hijas. Con tan solo usar el nombre de la clase padre en un modelo el objeto del sistema se define como la clase padre y las clases hijas.

En el ejemplo de la figura 37 la clase hija Manager tiene los atributos y operaciones que la clase padre Employee pero también tiene dos atributos propios de la clase, la otra clase hija Programmer además de los atributos heredados tiene dos atributos propios de esa clase. Se pueden usar los objetos de la clase Manager o Programmer en cualquier sitio que se necesite un objeto de la clase Employee.

Las asociaciones entre clase se da con la intervención de los miembros de una clase con los de otra clase. En UML las asociaciones son representadas por unas líneas que unen las clases involucradas en esa asociación, en la cual se puede añadir una nota información acerca de la asociación. Esto se lo puede ejemplarizar en la Figura 38 donde se observa las asociaciones de la clase Employee.

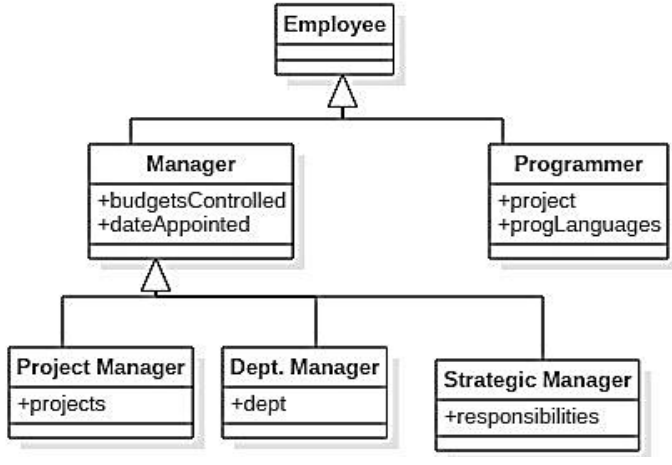



***Figura 37.*** Un ejemplo de jerarquía con generalización.

Una asociación es una relación entre clases, en ocasiones se usa en UML para señalar que un atributo de una clase asociada es un objeto asociado o que implementación del método de un objeto depende de la asociación del mismo. Una de las asociaciones más habituales es la agregación en la cual se puede manifestar que los objetos están compuestos por otros objetos.

#### 2.1.1.1. Objetos concurrentes.

Cuando un objeto manda un mensaje de "solicitud de servicio" a otro no es necesario esperar que la ejecución de este mensaje termine, puesto que el modelo general de interacción de objetos admite la ejecución concurrente como procesos en paralelo. Se pueden ejecutar estos objetos en diversas computadoras como objetos compartidos.

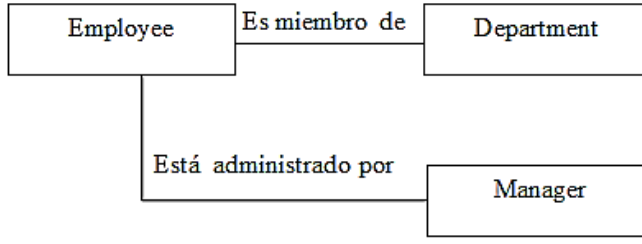



***Figura 38.*** Un ejemplo objetos concurrentes.

Algunos de los lenguajes de programación orientados a objetos poseen modelos de ejecución en serie, donde la solicitud de servicios a objetos se implementa de la misma manera que la de una función. En Java cuando objeto denominado the list aparece a partir de una clase normal se escribe:

theList.append (17)

Esta sintaxis invoca al método append que se encuentra relacionado con el objeto theList para agregar el elemento 17 a theList, hasta que la operación append no se haya terminado la ejecución del objeto que hace la llamada al método permanece interrumpida.

Java posee un mecanismo (hilos) que hace posible la creación de objetos que se ejecutan al mismo tiempo. En java los hilos son creados utilizando en la declaración de una clase a la clase Threat como cla-

se padre, deben incorporar un método definidos como ruc inicializado como run-time. Fácilmente se puede implementar un diseño orientado a objetos donde los objetos sean procesos concurrentes.
La implementación de los objetos concurrentes se la realiza de dos maneras.

- Servidores donde el objeto es implementado paralelamente con los métodos correspondientes a las operaciones declaradas de los objetos. Como contestación a un mensaje extremo los métodos empiezan su actividad y se ejecutan paralelamente con los métodos relacionados a otros objetos. Una vez terminada su operación, el objeto es interrumpido por él mismo esperando peticiones extra de los servicios.
- Objetos activos donde su estado se modifica por las operaciones internas ejecutadas. Estas operaciones no pueden ser interrumpidas debido a que el proceso que simboliza al objeto las ejecuta continuamente.

El uso de servidores en más frecuente en entornos distribuido donde los objetos tanto emisores como receptores se ejecutan en diferentes computadoras. Es recomendable diseñar el software de manera que el objeto que requiere un servicio no perder el tiempo esperando a que este se complete. Igualmente a los servidores se los puede usar en una sola computadora, donde a un servicio que es requerido por objetos diferentes le lleve cierto tiempo terminar (al momento de imprimir un documento).

Los objetos activos son usados cuando los objetos requieren actualizar su estado en intervalos específicos. Generalmente usado en sistemas de tiempo real donde los objetos esta relacionados con dispositivos de hardware que recolectan los datos del entono del sistema. Los métodos de los objetos dan acceso a la información a otros objetos.

La Figura 39 muestra la implementación en Java de un objeto activo. Esta clase de objetos es la representación de un transpondedor de un avión, el cual es el encargado de dar la información la posición actual del avión mediante un sistema de navegación vía satélite en respuesta a la petición del método givePosition. Capaz de responder a los mensajes de las computadoras para el control del tráfico aéreo. Es implementado como un hilo, donde un bucle continuo en el método run posee el código para calcular la posición del avión usando las señales del satélite.

```
class Transponder extends Thread {
        Position currentPosition;
        Coords c1,c2;
        Navigator theNavigator;

        public Position givePositon ()
        {
                Return currentPosition
        }
        public void run ()
        {
                While(true)
                {
                        C1= sat1.position();
                        C2= sat2.position();
                }
        }
}//Transponder
```

***Figura 39.*** Implementación de un objeto utilizando subprocesos de Java.

### 2.1.2. Un proceso de diseño orientado a objetos.

En esta parte se detalla el proceso de diseño orientado a objetos mediante un ejemplo del diseño de un sistema que se encarga del control de una estación meteorológica automatizada. Hay algunos métodos usados en el diseño orientado a objetos sin que exista uno mejor que otro, el proceso que se muestra en el ejemplo incorpora las actividades más comunes de varios de ellos.

Etapas que usa el proceso general en el diseño de un sistema.

1. Conocer y determinar el entorno y la forma de uso del sistema
2. Elaborar la estructura del sistema.
3. Reconocer los elementos relevantes del sistema
4. Elaborar los modelos de diseño.
5. Determinar las interfaces de los objetos.

Como en este proceso no hay actividades detalladas procesadas en secuencia no se muestra como un diagrama de proceso simple. Más bien las actividades se las pueden observar como entrelazadas que actúan entre sí. Se establecen los objetos y se determinan las interfaces com-

pletas o parcialmente en el instante de fijar la arquitectura del sistema. Cuando se crean los modelos de objetos estas definiciones se mejoran y lo que puede lograr que haya una modificación en la arquitectura del sistema

El diseño no es un proceso fácil de elaborar, se desarrolla con el planteamiento de soluciones puesto que dichas soluciones pueden ser modificadas al adquirir más información. Necesariamente en el momento que los problemas surjan se tendrá que retroceder para analizarlos, en algunas ocasiones se tendrá que inspeccionar los detalles de las soluciones para ver si funcionan, y en otras ocasiones tendrán que ser excluidos y ser integrados posteriormente en el proceso.

Para una mejor explicación de las actividades de este proceso se muestra un ejemplo del diseño de un sistema para crear mapas meteorológicos a partir de datos meteorológicos recolectados de forma automática. La especificación de los requerimientos para este sistema es muy extensa, pero se puede determinar la arquitectura de sistema mediante una descripción breve.

Esta explicación muestra que el sistema se refiere a la recolección de datos, a la integración de datos recolectados por diversas fuentes, también a archivar esos datos y a crear mapas meteorológicos. En la figura 40 se observa la arquitectura del sistema diseñada a partir de la descripción. Este sistema tiene una arquitectura de capas debido a que cada etapa es dependiente de la operación de la anterior.

En la Figura 40 para indicar que cada capa integra diferentes componentes se ha agregado el nombre de la capa en un símbolo de representación de paquetes en UML que se ha expresado como un subsistema. Para representar el conjunto de paquete y distintos objetos en UML se usan los paquetes.

En la figura 41 este modelo se ha ampliado para enseñar los componentes de cada subsistema que han sido obtenidos de la descripción del sistema. Los cuales continúan siendo abstractos. A partir de ahora, se toma en cuenta para el ejemplo el subsistema de la estación meteorológica.

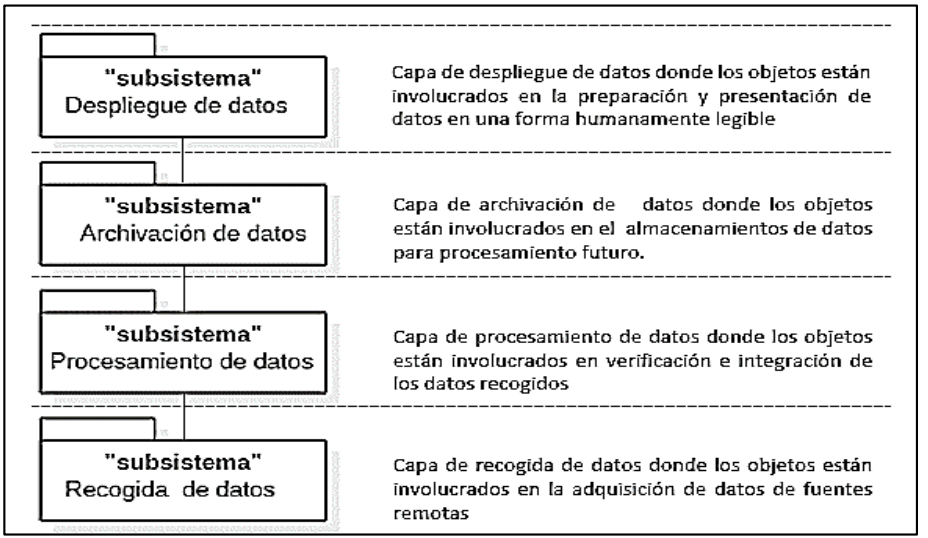



***Figura 40.*** Arquitectura de capas para un sistema de creación de mapas meteorológico.

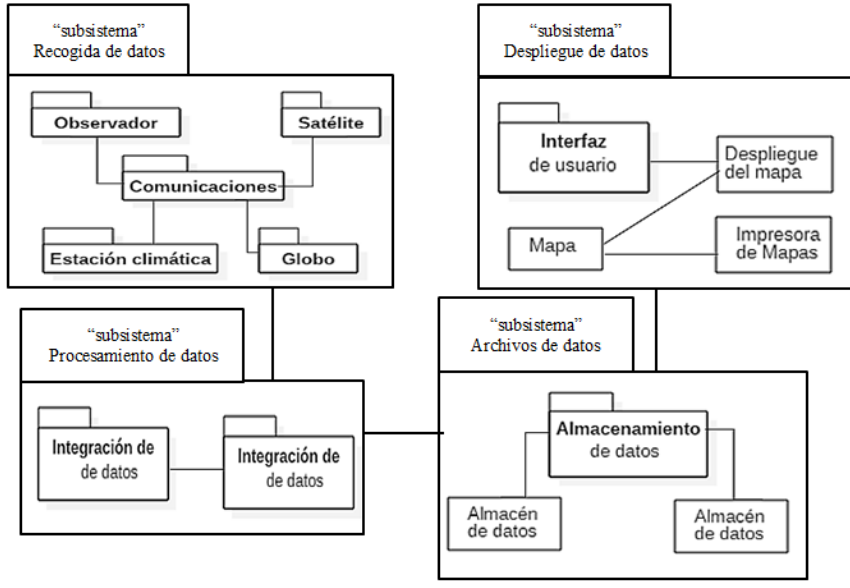



***Figura 41.*** Subsistemas en el ejemplo de mapas meteorológicos.

#### 2.1.2.1. Contexto del sistema y modelos de utilización.

En el desarrollo del diseño de software la primera etapa es la encargada de entender relación del software con el entorno exterior. Esta etapa nos ayuda a determinar cómo proporcionar la funcionalidad del sistema y estructurarlo para lograr una comunicación efectiva con su ambiente.

Modelos complementarios de la relación de un sistema con su entorno:

- Contexto del sistema, modelo estático encargado de explicar los demás sistemas de su ambiente.
- Modelo de utilización del sistema, modelo dinámico que explica la interacción del sistema con su ambiente.

Para simbolizar el contexto del sistema se usan asociaciones, en el que crea un diagrama de sencillo de bloques de la arquitectura de un sistema.

Esto se puede ampliar representado mediante paquetes UML un modelo del subsistema (Figura 41), lo cual muestra que el contexto de

la estación meteorológica se encuentra incluido en la recolección de datos. Además enseña los demás subsistemas que integran el sistema de “Trazo de mapas meteorológicos”.

El modelamiento de las interacciones de un sistema con su entorno no debe incorporar demasiados detalles de esa iteración, es decir, usar un planeamiento abstracto. En UML este modelamiento se lo puede desarrollar con diagramas de casos de uso, en el cual cada caso de uso es una interacción con el sistema y es representado por una elipse y cada actor del sistema (entidad externa) generalmente representada por un dibujo denominado “stick man” (hombres de palo). En el caso de uso sistema de estación meteorológica (Figura 42) el actor es el sistema de “procesamiento para los datos meteorológicos”, este caso de uso muestra la interacción entre la estación meteorológica con el actor, para iniciar, apagar dar información de los datos, calibrar y realizar pruebas de las herramientas.

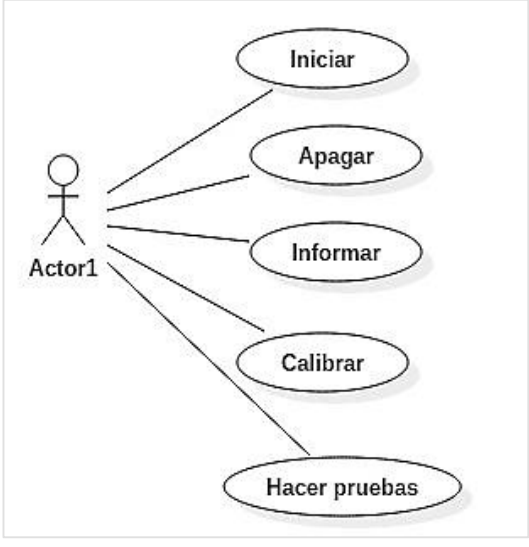



***Figura 42.*** Casos de usos de la estación meteorológica.

Para que los diseñadores puedan reconocer los objetos y comprender la función del mismo, se detalla los casos de uso usando un formulario estandarizado que permite especificar y establecer la información que se intercambia en los casos de uso.

En la figura 43 se muestra un ejemplo donde se describe el caso de uso “informar”.

| | |
|---|---|
| **Sistema** | **Estación meteorológica.** |
| **Caso alineo** | **Informar.** |
| **A d a m** | **Sistema de recogida de datos meteorológicos, estación meteorológica.** |
| **Datos** | **La estación meteorológica envía un resumen de tos datos meteorológicos recogidos de (os instrumentos en el periodo de acogida al sistema de recogida de datos meteorológicos. Los datos enviados son tos valores máximo, mínimo y promedio de las temperaturas del suelo y del aire; tas presiones del aire máxima, mínima y promedio; las velocidades del viento máxima, mínima y promedio; la turbia total y la dirección del viento tomada cada cinco minutos.** |
| **Estimulo** | **El sistema de recogida de datos meteorológicos establece un vínculo por medio del modem con la estación meteorológica solicita la transmisión de los datos.** |
| **Respuesta** | **Un resumen de los datos se envía al sistema de recogida de datos** |

***Figura 43.*** Un ejemplo de caso de uso “Informar”.

El detalle o descripción de los casos de uso permite determinar las operaciones y objetos que intervienen en el sistema.

Del detalle del caso de uso informar se puede observar que se necesitan objetos para representar las herramientas que recolectan los datos meteorológicos y además determinar operaciones que intervengan la solicitud y envío de datos.

### 2.1.2.2. Diseño de la arquitectura.

Cuando se hayan establecido las interacciones entre el sistema y su entorno, se puede establecer el diseño de la arquitectura tomando como origen información de dichas interacciones. Por lo tanto, es indispensable acoplarlo con el conocimiento general de los principios de diseño arquitectónico y con el conocimiento más detallado del dominio. El sencillo sistema de estación meteorológica automatizada posee una

arquitectura que puede ser representado como modelo en capas. En la figura 44 muestra este modelo en UML como un árbol de paquete. En el cual se ha usado para proveer información complementaria la notación en UML (cuadros de textos con una pestaña en la esquina).
Las capas del sistema de estación meteorológica son:

- La capa de interfaz, contiene todas las relaciones con otras partes del sistema y la distribución de las interfaces extremas.
- La capa de recogida de datos, encargada de dirigir la recolección y síntesis de los datos meteorológicos antes de enviarlos al sistema de mapas.
- La capa de instrumentos, compuesta por todas las herramientas usadas en la recolección de datos sobre las condiciones meteorológicas.

Se debe tratar de disgregar un sistema de manera que las arquitecturas sean lo más simple posible. Para un buen modelo arquitectónico se debe procurar no integrar más de siete entidades. Se puede detallar separadamente estas entidades, pero se debe mostrar la estructura de las entidades, como podemos observar en la siguiente figura:

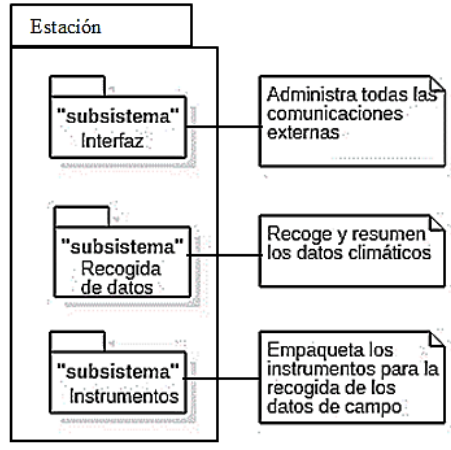



***Figura 44.*** La arquitectura de una estación meteorológica.

### 2.1.2.3. Identificación de objetos.

Con el desarrollo de las etapas anteriores del diseño de sistema ya se debe tener una idea los principales objetos del mismo, puesto que se

requiere por lo menos un objeto para cada nivel de la arquitectura. Algunos objetos pueden aparecer en el proceso de desarrollo del diseño, pero generalmente se debe buscar y registrar otros objetos importantes para el sistema.

Aunque el nombre de esta etapa del desarrollo del diseño sea "identificación de objetos", en la práctica en esta etapa se trata de identificar las clases. Como el diseño se crea a partir de estas clases, se tiene que modificar las clases identificadas en las etapas anteriores, ya que esta etapa ya se tiene mejor entendimiento sobre el diseño.

Existen diversas maneras de cómo identificar las clases.

- Analizar la descripción del caso de uso para identificar las clases, donde los sustantivos pueden ser tomados como los atributos, y los verbos las operaciones.
- Emplear entidades concretas (cosas) en el dominio de aplicación como carro, usar papeles como administrador, diseñador; eventos como una operación, ubicaciones como departamentos, unidades organizacionales como empresas; Esto se debe perfeccionar determinando estructuras de almacenamiento como parte de la solución, las cuales podrían solicitarse para ayudar a estos objetos.
- Analizar la conducta del sistema, en donde el diseñador debe entender el comportamiento del sistema, y asignarle comportamientos a varias partes del sistema, para entender quienes intervienen en el sistemas y que función realizan.
- Los integrantes del sistema que ejercen papeles relevantes en el sistema son identificados como objetos.
- Usar un análisis fundamentados en escenario, se determina y examinan distintos escenarios de la manera usar el sistema. Cada escenario debe ser analizado y el personal encargado del análisis debe establecer las operaciones, atributos y objetos requeridos para crear las clases. Hay un método seguro de análisis llamado tarjetas CRC, en el cual los analistas son los responsables de establecer las actividades de los objetos.

Estos enfoques colaboran en el origen del reconocimiento de objetos. Se usan diversas fuentes de conocimiento para revelar objetos y clases. Las operaciones y objetos establecidos en el análisis informal de los sistemas pueden ser usados como origen para el diseño. La in-

formación complementaria del conocimiento del dominio se usa para mejorar y ampliar los objetos iniciales.

Esta información se la recolecta de la documentación de requerimientos, las reuniones con los usuarios, y el análisis de los sistemas actuales, tal como se observa en la figura 45.

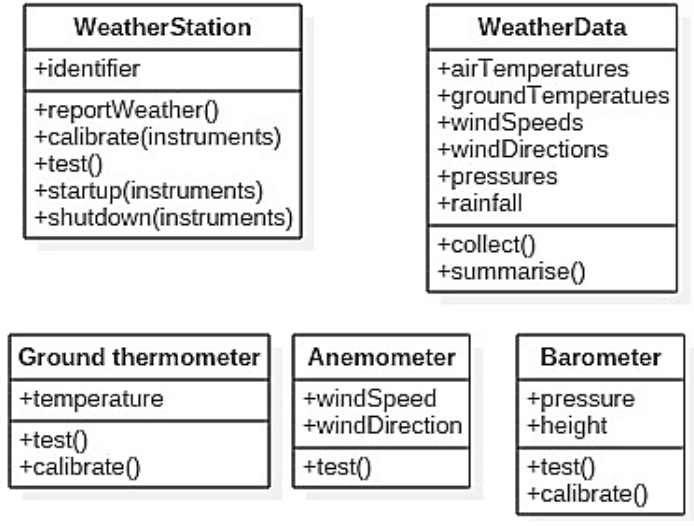



***Figura 45.*** Ejemplos de clases en el sistema de estación meteorológica

Estos objetos se relacionan con los diversos niveles de la arquitectura del sistema.

1. La clase WeatherStation suministra la interfaz básica de la estación meteorológica con su entorno. Para mostrar las operaciones utiliza una clase que encapsula todas las interacciones, también en otros diseños se puede utilizar varias clases para crear el diseño de la interfaz.

2. La clase de objetos WeatherData posee la síntesis de datos de los diferentes instrumentos en la estación meteorológica. Las operaciones involucradas en esta clase se encargan de la recolección de datos y la representación de los mismos.

3. Las clases GroundThermometer, Anemometer y Barom son enlazadas con los instrumentos del sistema. Sus operaciones controlan la parte de hardware tangible de ese sistema.

En esta parte del si stema se debe conocer correctamente el dominio de la aplicación para poder reconocer los servicios adicionales y objetos del sistema. Centrándonos en el caso del ejemplo, se sabe que las estaciones están ubicadas en lugares remotos, y poseen instrumentos que ocasionalmente su funcionamiento falla. Los errores de los instrumentos deben ser comunicados de forma automática. Por lo tanto son importantes las operaciones y atributos para comprobar el adecuado funcionamiento de los instrumentos. Como existen varias estaciones climatológicas remotas es necesario identificar los datos en cada estación, por lo que cada una debe poseer su propio identificador.

Se ha resuelto no crear objetos activos a los objetos relacionados con cada instrumento. En la clase WeatherData, la operación collect es la encargada de llamar a los objetos instrumentos para desarrollar las lecturas. Los sujetos activos son aquellos que incluyen su control personal, por tal razón cada instrumento determina cuando realizar las lecturas. El inconveniente que nos presenta esto que si se modifican los tiempos de la recogida de datos de las estaciones o la forma de la recolección, necesariamente deben incluirse nuevas clases. Si se hace que los objetos desarrollen su lectura bajo petición, cualquier modificación en la táctica de recolección se puede desarrollar sin necesidad de cambiar los objetos relacionados con los instrumentos.

#### 2.1.2.4. Modelos de diseño.

Son encargados de mostrar los objetos o clases de un sistema y las relaciones entre las mismas. Estos modelos son el enlace de los requerimientos de los sistemas y la implementación del mismo. Deben tener un contenido bien detallado para que los programadores tomen buenas decisiones de implementación, pero a pesar de esto se debe tomar en cuenta que el detalle no necesario puede ocultar la relación entre la implementación y los requerimientos del sistema.

Para solucionar esta situación se puede desarrollar varios modelos en distintos niveles de detalle. Si hay una relación cerca entre miembros del proyecto del sistema (ingenieros de requerimientos, diseñadores y programadores) se desarrollan modelos abstractos. Pero si la relación entre ellos es indirecta (por ejemplo, cuando el sistema es desarrollado por una parte del equipo y es implementado por otros), se necesitan modelo del sistema más detallado.

Es importante decidir qué modelo de diseño implementar, no se puede escoger cualquier modelo para el diseño, esto depende únicamente del tipo de sistema que se esté desarrollando. El diseño de un sistema de procesamientos secuencial de datos es distinto a un sistema empleado en tiempo real, por lo cual el modelo de diseño a implementar no es el mismo. Sin embargo hay sistema donde es necesario implementar todos los elementos. Disminuir el número de modelos en la implementación del diseño reduce costos y tiempo.

Tipos de modelo de diseño:

1. Modelos estáticos, encargados de detallar la estructura estática del sistema en forma de clases con sus relaciones. Las más relevantes relaciones que se detallan en esta etapa son de composición y generalización.

2. Modelos dinámicos, encargados de detallar la estructura dinámica del sistema y de manifestar las interacciones de los objetos del mismo. En dichas interacciones se debe incorporar la sucesión de servicios gestionados y la manera en que el estado del sistema se vincula las interacciones de los objetos.

Para la documentación del diseño UML facilita 12 modelos estáticos y dinámicos.

La explicación de todos esos modelos se hace un poco extensa, por tal razón en esta sección se detalla los modelos usados en el ejemplo (sistema de la estación climatológica), dichos modelos se detallan a continuación:

- Modelos de subsistemas, son modelos estáticos que enseñan los conjuntos lógicos de objetos en subsistemas congruentes. Son representados usando el diseño de los diagramas de clase, donde cada subsistema es presentado como un paquete.
- Modelos de secuencia, son modelos dinámicos donde se puede observar las interacciones de los objetos. Pueden ser representados como diagramas de colaboración o como una secuencia UML.
- Modelos de máquinas de estado, en donde se puede observar el cambio de estado de los objetos como contestación a los eventos. En UML pueden ser representados por diagramas de estado.

En la figura 46 se detallan los objetos de los subsistemas de la estación

meteorológica. Para poder establecer los servicios adicionales y los objetos se necesita tener conocimiento sobre el dominio de la aplicación. En este modelo se pueden observar varias asociaciones. Por ejemplo, el objeto CommsControler está relacionado con el objeto WeatherStation, y éste con el paquete Data collection. Esto nos quiere decir que el objeto está relacionado con uno o más objetos de este paquete. El detalle de los conjuntos lógicos del sistema se logra con los modelos de paquetes y clases.

Un modelo lógico de subsistemas mediante la agrupación de objetos puede estar estructurado el diseño del sistema de forma lógica. En la figura 41 se detalla un ejemplo de este modelo (subsistemas del sistema de mapas meteorológicos). En UML los paquetes incluyen las construcciones de encapsulamiento y no se manifiesta claramente sobre las entidades del sistema. Por otro lado pueden verse como construcciones de estructuras, así como las librerías de Java.

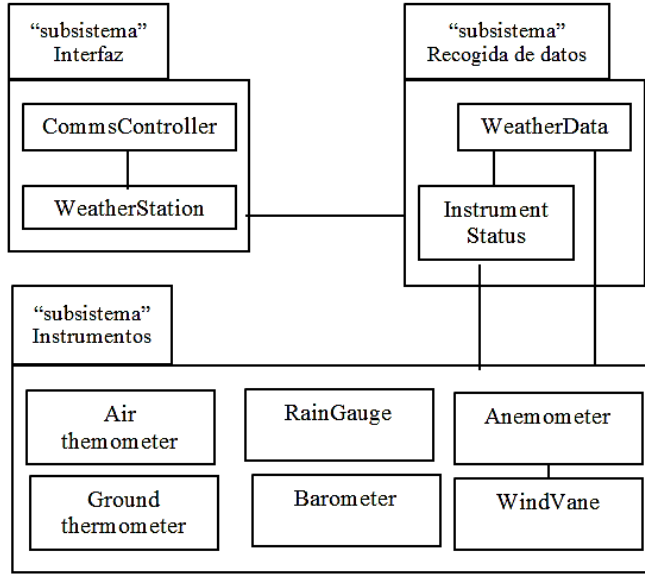



***Figura 46.*** Subsistema de una estación meteorológica.

Los modelos de secuencia son los encargados de informar la sucesión de interacciones entre los objetos. La figura 47 detalla un ejemplo donde se observa las operaciones implicadas en la recolección de datos del sistema de esta meteorológica mediante un modelo de secuencia:

1. Los objetos que participan en la interacción del sistema, se encuentran ubicados horizontalmente de forma ordenada con una línea vertical que permiten la relación de cada objeto.
2. El tiempo es representado de forma vertical con líneas punteadas, que avanza hacia abajo esto facilita la lectura de la secuencia de operaciones.
3. Las flechas etiquetadas que relacionan las líneas que simbolizan el tiempo representan las interacciones entre objetos. Estas flechas muestran los mensajes o eventos que son esenciales para la interacción.
4. Para representar el tiempo que un objeto posee el control del sistema se ubican cuadros pequeños sobre la línea de vida del objeto. El control comienza en la parte superior del rectángulo y finaliza en la parte inferior del mismo. Si hay una dependencia de llamadas, el control no finaliza hasta que se haya concluido el último retorno al método inicial.

En el momento de documentar un diseño, se debe hacer un modelo para cada interacción un modelo de secuencia. Se debe crear un modelo de secuencia por cada de uso.

El ejemplo que se muestra en la figura 47 es un diagrama de secuencia de interacciones que representa, la solicitud de datos a la estación meteorológica del sistema extremo de trazado de mapas. La lectura de este diagrama se la puede realizar de la siguiente manera:

Los diagramas de secuencia son de gran utilidad, ya que permite modelar el comportamiento de un conjunto de objetos y sintetizar el comportamiento de un objeto como como respuesta a los distintos mensajes que se pueden desarrollar. Esto se logra con el uso de un modelo de máquina de estado en el que se puede observar cómo se modifica el estado de la instancia de un objeto según los mensajes que se reciban. Los modelos de máquina de estado en UML son representados mediantes diagramas de estado.

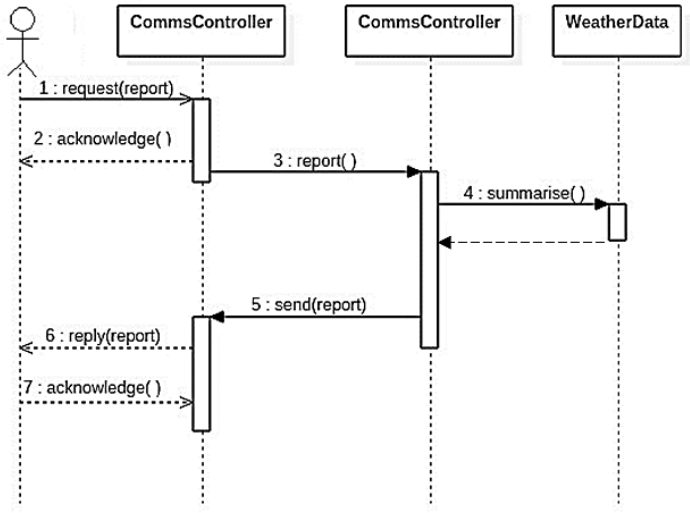



***Figura 47.*** Secuencia de las operaciones de recogida de datos.

En la figura 48 se muestra un diagrama de estado donde se representa respuesta a la petición de distintos servicios.
El diagrama de estado del ejemplo se lo puede leer de la siguiente manera:

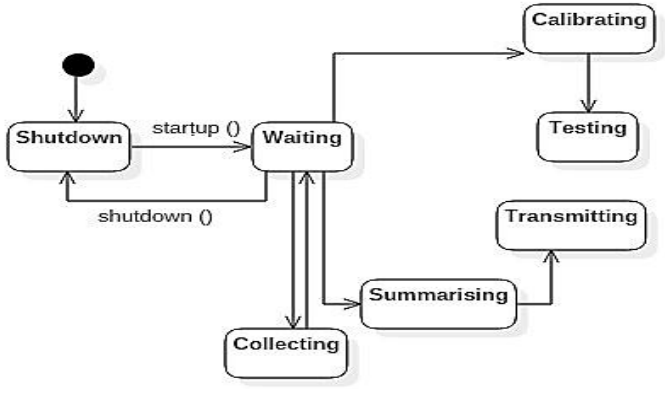



***Figura 48.*** Diagrama de estado para WeatherStation.

Por lo general, no es indispensable crear un diagrama de estado para cada objeto definido en el sistema. Ya que puede haber objetos sencillos, y un diagrama de este tipo no sería de este tipo no sería de gran utilidad para los programadores. .

#### 2.1.2.5. Especificación de la interfaz de los objetos.

Es de gran importancia en el proceso de diseño determinar las interfaces del sistema. Para que los objetos y demás componentes se puedan diseñar paralelamente es imprescindible determinar las interfaces.
El personal encargado de diseño debe eludir la información que representa la interfaz, en cambio se debe proporcionar las operaciones de los objetos para acceder y actualizar datos. Esto permite que se puedan realizar cambios en la interfaz sin perjudicar a los objetos que usan esos atributos.
El diseño de interfaces de objetos, se trata de la especificación de la interfaz para un objeto o grupos del mismo. Es decir determinar las firmas y semántica de los servicios proporcionados por los objetos o por los grupos de los mismos. En UML las interfaces de objetos se determinan con un diseño similar a los diagramas de clases, aunque en la interfaz no hay atributos.
Se considera enfoque alternativo cuando se determinar la interfaz con el uso de un lenguaje de programación. En la figura 49 se detalla un ejemplo de una interfaz en java. Este enfoque es más acertado ya que al compilar se pueden observar los errores e incoherencias en
Si las interfaces son más complejas, este enfoque es más efectivo debido a que las herramientas de verificación de sintaxis en el compilador se utilizan para descubrir errores e incongruencias en la especificación de la interfaz.

### 2.1.3. Evolución del diseño.

La ventaja de usar un enfoque orientado a objetos para el desarrollo del diseño es que las modificaciones que se hagan a los detalles internos de un objeto no perjudican a otros objetos, esto se logra debido a que los objetos no están muy acoplados. En la figura 50 se demuestra la robustez del enfoque orientado a objetos se detalla el siguiente ejemplo:

```
Interface Weatherstation
{

        public void Weatherstation O.;

        public void startup O."
        public void startup (instrument i);

        public void shuldown O;
        public void shuldown (instrument i);

        public void Reportweather ();

        public void test O;
        public void test (instrument i);

        public void calibrate (instrument i);

        public int getiD O;

}// Weatherstation
```

***Figura 49.*** Descripción de Java de la interfaz de la estación meteorológica

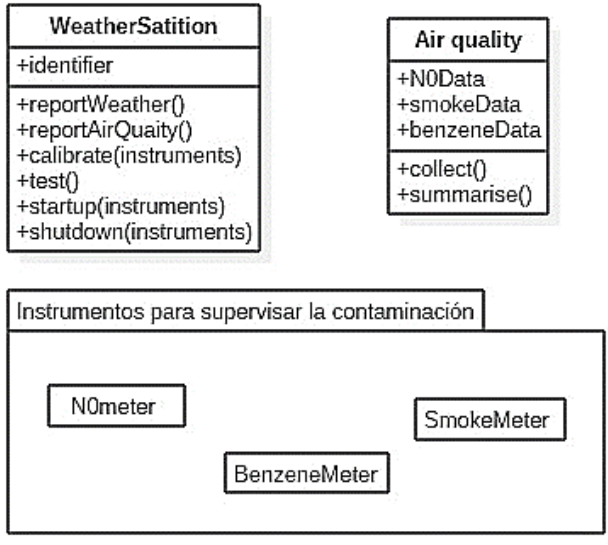



***Figura 50.*** Nuevos objetos para ayudar a la supervisión de la contaminación.

## 2.2.Diseño de software de tiempo real.

Los ordenadores son utilizados inspeccionar varios sistemas desde artefactos domésticos simples a grandes fábricas; estos recíprocamente se relacionan con periféricos hardware. El software de estos sistemas es de tiempo real embebido que debería responder a sucesos desarrollados por el hardware y emitir símbolos como respuestas a los mismos. Este software está sumergido en un sistema de hardware más amplio, y debe responder los acontecimientos en tiempo real.
Los sistemas de tiempo real embebidos, no son iguales a otro sistema software. Su función es correcta cuando lo hace en poco espacio de tiempo. Se podría conceptuar un sistema de tiempo real de la siguiente manera:

Es un sistema software de perfecta marcha, si los logros obtenidos del mismo se dan en el intervalo de tiempo que producen estos resultados. Un software o sistema de tiempo real blando, pierde valor si la solución no se elabora en alianza con las demandas momentáneas precisadas. Para que un sistema de tiempo real duro no sea correcto la conclusión no se hace en convenio con la precisión del momento.
Una solución oportuna es un elemento de primer orden en sistemas embebidos, según sea el caso dicha respuesta no debe ser tan veloz. Un ejemplo, el sistema de bomba de insulina es un sistema embebido; al necesitarse saber el nivel de glucosa periódicamente, deja de ser imprescindible una respuesta ágil a eventualidades exageradas. Sin embargo se usan diferentes ejemplos para enseñar los cimientos de diseño de los sistemas de tiempo real.

Una manera de examinar un sistema de tiempo real es como un sistema de impulso/solución. Por tanto definiremos el comportamiento de un sistema de tiempo real, detallando los impulsos captados por el sistema, las soluciones asociadas y el momento en que dichas sentencias deben elaborarse.
A estas dos clases pueden corresponder los estímulos:

- Estímulos periódicos. Se dan a espacios de tiempo predecibles. Ejemplo, el sistema observa un sensor cada 50 milisegundos y ejecuta una labor o respuesta de acuerdo del valor de ese sensor o estímulo.
- Estímulos aperiódicos. Suceden de manera no regular. Fre-

cuentemente son estimulados, usando la mecánica de interrupción de la computadora. Ejemplo de este estímulo sería la interrupción indicando que una transferencia de E/S ha concluido y los datos están a disposición en un búfer.

El modelo sensor-sistema que actúa en un sistema de tiempo real embebido se ilustra en la figura 51 que encontramos a continuación.

El sistema de tiempo real responde a impulsos que se dan en tiempos distintos. Por lo cual organizan su estructura para enseguida captar el impulso, se transfiera al manejador oportuno; el mismo no es útil en programas de secuencia. Lo sencillo de este sistema operativo es su accesibilidad por medio del sistema de soporte en tiempo de realización.

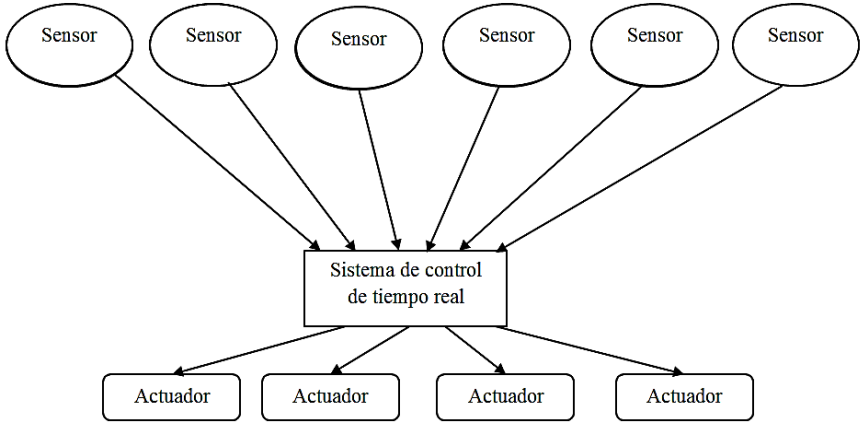



***Figura 51.*** Modelo general de un sistema en tiempo real.

Lo común en este tipo de estímulo-respuesta en un sistema de tiempo real nos lleva a un prototipo arquitectónico genérico abstracto que contiene tres modelos de procesos. Este prototipo permite la recolección acelerada de datos desde el sensor y logra que procese y la sentencia asociada al actuador se ejecute más adelante.

La arquitectura común puede instanciarse a algunas arquitecturas de aplicaciones distintas que agrandan el conjunto. En este apartado se incorporan dos arquitecturas de aplicaciones: arquitectura para sistemas de monitoreo y control y arquitectura para sistemas de obtención de datos.

En un software de tiempo real duros todavía se programan a veces en ensamblador para cumplir estrechos espacios de tiempo (deadline). La facilidad del uso de un lenguaje de programación de sistemas de bajo nivel como C, es permitir la creación de eficientes programas; pero estos lenguajes no contienen construcciones capaces de soportar la concurrencia de medios divididos, aumentados por medio de llamadas sistema operativo de tiempo real, no comprobado por el compilador.
Pocos años atrás se viene haciendo una gran labor con fines de emplear Java para elaborar sistemas de tiempo real:

1. No se puede determinar el momento de tiempo en que los hilos se deberían ejecutar.
2. No se puede controlar la recolección de basura que puede iniciarse en cualquier tiempo; en tanto la conducta temporal de los hilos no se puede predecir.
3. Sería imposible averiguar los portes de las colas agrupadas con medios divididos.
4. El equipamiento de la Máquina Virtual de Java es diferente de una a otra computadora, pudiendo tener conductas distintas en diferentes tiempos.
5. El lenguaje no deja hacer un listado de la memoria o del procesador en temporal de acción.
6. No existe un acceso estandarizado al hardware del sistema.

### 2.2.1. Diseño del sistema.

Para el diseño de un sistema en tiempo real primeramente se debe determinar que parte del sistema se debe implementar en el software y que parte en el hardware.

El proceso de diseño de un software de tiempo real empieza determinando un modelo abstracto que se va dividiéndose y desarrollando en una sucesión de etapas, aunque esto no es de utilidad para la mayoría de los sistemas. Para no limitar la flexibilidad del diseñador del sistema es necesario que se establezcan como se va a manejar el software y hardware del sistema.

Etapas para determinar los eventos en los procesos de diseño de un sistema.

1. Determinar los estímulos y respuestas asociadas que se deben desarrollar en el sistema.

2. Determinar las restricciones temporales del sistema aplicadas en el proceso y estimulo de respuesta.
3. Considerando las restricciones del sistema, costo del desarrollo y la experiencia del grupo encargado del desarrollo se debe elegir una plataforma para el sistema operativo y hardware que se vaya a utilizar.
4. Relacionar los procesos, los tipos de estímulo y la respuesta, e integrar el proceso de estímulos y la respuesta a varios procesos concurrentes.
5. Crear algoritmos para proporcionar una indicación de la cantidad de procesamiento requerido y del tiempo necesario para terminar dicho procesamiento.
6. Crear una planificación de los procesos involucrados en el desarrollo del sistema.

No siempre las actividades se desarrollarán de la misma forma, si no que cambiarán el orden según el tipo de sistema que está elaborando y los requerimientos de su proceso.

Los procesos de un sistema de tiempo real deben estar coordinados para asegurar la exclusión mutua de recursos compartidos. De modo que si un proceso cambia un recurso compartido, no se les debería permitir a los demás procesos también modificar ese recurso. Teniendo establecida una plataforma de ejecución, diseñada la arquitectura para el proceso y haber escogido una política de planificación, se debe verificar que el sistema cumple sus requerimientos temporales. Esto se puede realizar a través del análisis estático del sistema.

Como el análisis temporal para un sistema de tiempo real es dificultoso, los diseñadores tienen que deducir sobre la probabilidad de la ocurrencia de esto estímulos, en la figura 52 observamos este proceso.

En los sistemas de tiempo real duro no se puede emplear un proceso de desarrollo orientado a objetos, ya que estos sistemas deben satisfacer sus restricciones temporales.

#### 2.2.1.1. Modelado de sistemas de tiempo real.

Los sistemas de tiempo real deben responder a eventos, los cuales a veces hacen que el sistema modifique su estado. Por esta razón, para modelar sistemas de tiempo real un modelo de máquinas de estado.
Los modelos de máquina de estados son una parte importante de los métodos usados en el diseño de sistemas de tiempo real ya que mediante estos modelos se puede representar el diseño de un sistema. UML permite crear modelos de estados basados en diagramas de estado.

Un modelo de estados de un sistema sabe que cuando se recibe un estímulo se puede producir una transición que implica el cambio de estado. Por ejemplo en la figura 53 observamos modelo de máquina de estados de una bomba de petróleo, donde el sistema que controla una puerta eléctrica puede cambiar su estado, de "puerta abierta" a "puerta cerrada" cuando se recibe una orden (el estímulo) del usuario (ordenador).

### 2.2.2. Sistemas operativos de tiempo real.

Los sistemas de tiempo real trabajan en conjunto con los sistemas operativos de tiempo real, el cual gestiona los procesos y agrega recursos en un sistema de tiempo real y además detienen los procesos para que los estímulos puedan ser manejados y asigna memoria y recursos del procesador.

Los componentes de los sistemas operativos de tiempo real (RTOS) dependen del tamaño y del grado de complejidad del sistema, los sistemas más sencillos incluyen los siguientes componentes:

1. Un reloj de tiempo real. Provee información para programar los procesos de forma constante.
2. Un manejador de interrupciones. Realiza las solicitudes no periódicas de los servicios.
3. Un planificador. Encargado de revisar los procesos que pueden ser ejecutados y elegir uno de ellos para su ejecución.
4. Un gestor de recursos. Es el encargado de asignar la memoria adecuada y los recursos del procesador.
5. Un despachador. Encargado de iniciar la ejecución del proceso

Los sistemas operativos de tiempo real implementados en grandes sistemas, pueden obtener facilidades adicionales, tales como gestión de

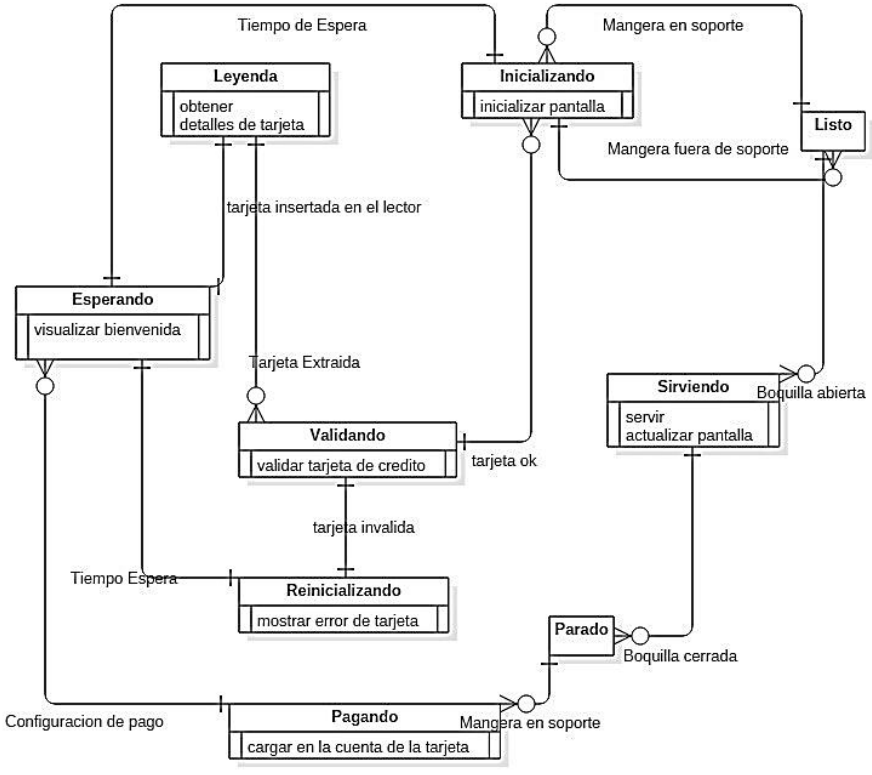



***Figura 53.*** Modelo de máquina de estados de una bomba de petróleo.

almacenamiento en disco de fallos, encargados de detectar e informar los fallos encontrados en el sistema y un gestor de configuraciones.

**2.2.2.1. Gestión de procesos.**

En los sistemas de tiempo real los procesos de manejo de eventos deben ser elaborados a tiempo para su ejecución y así detectar el evento, y se deben asignar los recursos necesarios del procesador para satisfacer su periodo de tiempo.

El gestor de procesos en los sistemas de tiempo real se encarga de escoger los procesos para su ejecución, delegar el procesador y recursos de memoria, e iniciar y detener la ejecución de un proceso sobre un procesador.

Para algunos estímulos, como los relacionados con algunos eventos excepcionales, es fundamental que su procesamiento sea determinado

dentro del periodo de tiempo especificado. En los sistemas operativos de tiempo real (RTOS) deberían tener la capacidad de gestionar al menos dos niveles de preferencia para los procesos del sistema:

1. Nivel de interrupción, tiene la prioridad más alta. Es asignado a procesos que requieren una respuesta rápida.
2. Nivel de reloj, es asignado a los procesos periódicos. Se les puede asignar un nivel más de prioridad procesos que se ejecutan en un segundo plano, es decir que no poseen un periodo límite de culminación.

En cada nivel de prioridad, se le puede determinar a los diferentes tipos de proceso distintas prioridades.

Los procesos periódicos son procesos que deben ejecutarse a intervalos de tiempo predefinidos para la adquisición de datos y el control de los actuadores. En la mayoría de los sistemas de tiempo real, habrá varios tipos de procesos periódicos. Éstos tendrán diferentes periodos (el tiempo transcurrido entre ejecuciones del proceso), tiempos de ejecución y plazos de tiempo (el tiempo en el cual se debe completar el procesamiento). Utilizando los requerimientos temporales especificados en el programa de la aplicación, el RTOS ordena la ejecución de los procesos periódicos para que todos ellos puedan cumplir sus plazos de tiempo.

El planificador examina la lista de procesos periódicos y selecciona un proceso para su ejecución. La elección depende de la prioridad de los procesos, de los periodos de los procesos, de los tiempos de ejecución esperados y de los plazos de tiempo de los procesos listos para ejecución. Algunas veces, dos procesos con diferentes plazos de tiempo deberían ejecutarse en el mismo tic del reloj. En tal situación, un proceso debe retrasarse de forma que su plazo de tiempo todavía pueda cumplirse.

Los procesos que tienen que responder a eventos asíncronos son normalmente conducidos por interrupciones. El mecanismo de interrupciones de la computadora hace que el control se transfiera a una posición de memoria predeterminada. Esta posición contiene una instrucción para saltar a una rutina de servicio de interrupciones sencilla y más rápida. La rutina de servicio de interrupciones primero inhabilita las interrupciones para evitar ser interrumpida.

Seguidamente, encuentra la causa de la interrupción e inicia, con

la prioridad más alta, un proceso que maneja los estímulos que provocan la interrupción. En algunos sistemas de adquisición de datos de alta velocidad, el manejador de interrupciones guarda los datos marcados por la interrupción para su procesamiento posterior. A continuación, las interrupciones se habilitan de nuevo y el control se devuelve al sistema operativo.

El planificador del proceso para determinar el orden de la ejecución de los procesos incluye políticas de planificación.

Hay dos estrategias fundamentales de planificación:

1. Planificador sin reemplazo. Cuando un proceso ha sido planificado para su ejecución, su ejecución se realiza hasta el final hasta cuando se bloquee por alguna razón. Esto puede ocasionar conflictos cuando hay procesos con distintas prioridades y un proceso de prioridad alta debe esperar a que culmine el proceso de menor prioridad.
2. Planificación con reemplazo. Se puede suspender la ejecución si un proceso con mayor prioridad necesita usar el procesador, reemplazando la ejecución del proceso de menor prioridad por el de mayor.

Para estas estrategias se han elaborado algoritmos de planificación, como la planificación de round-robin, donde los procesos son ejecutados por turnos, la planificación de frecuencia monótona, en la que la prioridad la tiene el proceso más corto y su estrategia trata de ejecutar primero el proceso con el periodo de tiempo más corto.

La información para ejecutar los procesos es enviada por el gestor de recursos. Cuya información es enviada memoria y en un sistema multiprocesador el proceso es enviado a un procesador.

Luego los procesos se sitúan en una lista de procesos listos para la ejecución. Cuando un procesador culmina la ejecución del proceso y vuelve estar disponible se llama al despachador, el cual es el encargado de examinar las listas de procesos preparados, para buscar y ejecutar el próximo proceso a ejecutarse.

### 2.2.3. Sistemas de monitorización y control.

Son una clase relevante del sistema de tiempo real. Encargados de comprobar que proporcionan información del ambiente del sistema y dependiendo de la lectura realizar las acciones. Estos sistemas desa-

rrollan una acción cuando se localiza un valor peculiar del sensor. Los sistemas de control controlan continuamente los actuadores hardware dependiendo del valor de los sensores relacionados.

Cada modelo de sensor que se está monitoreando posee su propio proceso de monitorización. Un proceso de monitorización se encarga de recoger e integra los datos antes de enviarlos a un proceso encargado del control, y toma decisiones fundamentos en esos datos y envía los comandos de control necesarios a los procesos de control del equipo.
En sistemas simples se pueden constituir en un solo proceso las responsabilidades de monitorización y control. Hay dos procesos que también pueden integrarse en sistemas de monitorización y control. Los cuales son: proceso de pruebas encargados de ejecutar programas de test del hardware y un proceso de panel de control que tramita los paneles de control del sistema.

El paso posterior en el proceso de diseño es examinar las restricciones eventuales relacionadas a cada estímulo y su correspondiente respuesta. Generalmente se debe numerar las restricciones eventuales para cada tipo de sensor de manera independiente. Gestionándolas de manera independiente, permitir futuros cambios y resulta más fácil calcular el número de veces que el proceso de control tiene que ejecutarse cada segundo.

El número de sensores que tienen que ser consultados y los requerimientos temporales del sistema se utilizan para calcular la periodicidad con la que cada proceso tiene que ser planificado.
Una vez que ha sido definida la arquitectura de los procesos del sistema, deberían diseñarse los algoritmos para el procesamiento de los estímulos y la generación de respuestas, esta etapa de diseño detallado es necesaria al principio del proceso de diseño para asegurar que el sistema pueda cumplir sus restricciones de tiempo especificadas. Si los algoritmos asociados son complejos, se pueden requerir cambios en las restricciones de tiempo. Sin embargo, a menos que se requiera el procesamiento de señales, los algoritmos de sistemas de tiempo real son a menudo bastante sencillos. Éstos solamente pueden requerir la comprobación de una posición de memoria, realizar algunos cálculos sencillos o emitir una señal.

En el proceso de diseño el último paso es el diseño de un sistema de planificación que garantice que un dicho proceso será planificado

para cumplir con sus periodos de tiempo.

Un ejemplo de un sistema de control sería un sistema que controla la calefacción de un edificio. Se encarga monitorizar los sensores de temperatura en diferentes partes del edificio, además está encargado de apagar y encender la calefacción dependiendo de la temperatura actual y la fijada en el termostato de dicha estancia. Además el termostato controla generador de calor del sistema.

### 2.2.4. Sistemas de adquisición de datos.

Se encargan de recoger los datos de sensores para su siguiente procesamiento y análisis. Son usados en situaciones en las que los sensores han recolectado extensas cantidades de datos del ambiente del sistema. Generalmente estos sistemas se usan en el control de procesos en los que los procesos físicos, tales como una reacción química, suceden velozmente.

En los sistemas de adquisición de datos, los sensores pueden estar generando datos muy velozmente, y el problema principal de esto es asegurar que cambie el valor del sensor antes de que la lectura del sensor sea recolectada.

La principal característica de la arquitectura de los sistemas de adquisición de datos es que tienen tres procesos relacionados en cada grupo de sensores: el proceso del sensor que relaciona con el sensor y cambia datos analógicos a valores digitales si se requiere, un proceso búfer, y un proceso que agota los datos y desarrolla un procesamiento adicional.

El número de sensores de un conjunto depende de la velocidad a la que recibe los datos de su entorno, los sensores pueden ser de distintos tipos.

## 2.3. Diseño de interfaces de usuario

El boceto de sistemas informáticos comprende diversas acciones que van desde el boceto de hardware incluso el de la interfaz de usuario. Sin embargo varios expertos a menudo trabajan en el boceto de hardware y en el boceto detallado de páginas web, regularmente sólo las organizaciones magnas utilizan creadores expertos de interfaces para sus aplicaciones software. Por tanto, los especialistas de software a me-

nudo deben tomar el compromiso de bosquejar la interfaz de usuario, así tanto del boceto del software que realiza esa interfaz.

Aunque los programadores y diseñadores de software son interesados competentes en la tecnología manipulada en la ejecución de las interfaces, como las clases Swing de Java o XHTML, las interfaces de usuario que desenvuelven muy seguido son poco seductoras e inadecuadas para sus interesados objetivos. Se examinará, por lo tanto, el transcurso del diseño de interfaces de usuario en lugar del software que realiza estas interfaces. A causa de las restricciones de espacio, sólo se tomaran en cuenta las interfaces gráficas de usuario. No se consideraran las interfaces que necesiten un método especial (aunque tal vez muy sencillo) de representación como las de televisores, telefonía móvil, copiadoras, reproductores de DVD y máquinas de fax. Normalmente, a partir de aquí podemos dar un inicio, por lo cual para obtener mayor detalle sobre el diseño de interfaces de usuario se sugiere los textos de Dix y otros, Shneidermn y Weiss.

La parte esencial del desarrollo de diseño general del software es tener un diseño cuidadoso de la interfaz de usuario. Si un sistema software debe lograr su máximo potencial, es esencial que su interfaz de usuario sea bosquejada para concordar con las habilidades, expectativas y experiencia de sus interesados previstos. Un buen bosquejo de la interfaz de usuario es crítico para la confidencialidad del sistema. Muchos de los citados «errores de usuario» son producidos por el hecho de que las interfaces de usuario no analizan las habilidades de los usuarios reales y su ambiente de trabajo. Una interfaz de usuario mal diseñada simboliza que los usuarios posiblemente no podrán entrar a algunos detalles del sistema, realizarán errores y apreciarán que el sistema les dificulta en vez de apoyarlos a lograr cualquier objetivo para el que usan el sistema.

Al momento de tomar decisiones en el diseño de las interfaces de usuario, deben tenerse en cuenta las capacidades físicas y mentales de las personas que manejarán el software. Las restricciones de espacio no consiente tratar aquí en detalle los factores humanos, pero varios factores significativos que deben considerarse son los siguientes:

1. La memoria a corto plazo de las personas: logramos recordar repentinamente cerca de siete elementos de información. Por lo tanto, si a los usuarios se les presenta excesiva información en

similar tiempo, es posible que no puedan equipararla.

2. Cualesquiera comete errores, principalmente cuando tenemos que manipular excesiva información o estamos estresados. Cuando los sistemas fracasan y expresan mensajes de aviso y alarmas, frecuentemente aumentan el estrés de los usuarios, aumentando así la posibilidad de que realicen errores.
3. Tenemos un extenso rango de capacidades físicas. Unas personas ven y escuchan mejor que las demás, otras son daltónicas, y otras son mejores en manipulaciones físicas. No se debe diseñar para las propias capacidades y presumir que todos los otros usuarios serán aptos de adaptarse.
4. Poseemos desiguales preferencias de interacción. A ciertas personas les gusta trabajar con imágenes, a otras con texto. La administración directa es natural para ciertas personas, pero otras eligen un estilo de interacción basado en expresar comandos al sistema.

Estos elementos humanos son la base para las iniciaciones de diseño que se muestran en la Figura 2.20. Estas iniciaciones generales se emplean a todos los diseños de interfaces de usuario y regularmente se deben instanciar como directrices de diseño más precisas para ordenaciones o tipologías de sistema específicos. Las iniciaciones de diseño de las interfaces de usuario son tratados con mayor referencia por Dix y otros. Shneiderman da una lista más extensa de directrices más concretas para el diseño de las interfaces de usuario.

Según el principio de familiaridad del usuario sugiere que los usuarios no deben ser obligatorios a acomodarse a una interfaz sólo porque sea provechoso implementarla. La interfaz debe manejar términos familiares para los interesados, y los objetos que el sistema manipula deben estar claramente relacionados con el ambiente de trabajo del usuario. Por ejemplo, si un sistema se diseña para ser manipulado por examinadores del tráfico aéreo, los objetos deben ser aviones, recorridos de vuelo, aerofaros, etcétera. Los procedimientos asociados podrían ser aumentar o reducir la aceleración del avión, ajustar la ubicación del avión y cambiar de altura. La añadidura subyacente de la interfaz en lo que se refiere a estructuras y archivos de datos se debe esconder al usuario final.

El principio de uniformidad de la interfaz de usuario expresa que

los menús y comandos del sistema deben poseer igual formato, los parámetros deben pasarse a cualquiera de los comandos de igual forma, y la puntuación de los comandos debe ser aparecida. Las interfaces uniformes disminuyen el tiempo de enseñanza del usuario. Por lo cual, el conocimiento asimilado en un comando o aplicación es aplicable en otras partes del sistema o en aplicaciones afines.

La uniformidad de la interfaz a lo extenso de las aplicaciones también es significativa. En lo viable, los comandos con significados equivalentes en aplicaciones desiguales se deben expresar de igual manera. Muy seguido los errores se originan cuando el semejante comando del teclado, como Control+b, representa cosas incomparables en sistemas diferentes. Por ejemplo, en el procesador de textos que utilizo habitualmente, Control+b representa colocar en negrita el texto, pero en los programas gráficos que utilizo para trazar diagramas, Control+b simboliza poner el objeto marcado detrás de otro objeto. Yo ejecuto errores cuando los manejo al mismo tiempo y muy seguido trato de poner en negrita el texto en un diagrama utilizando esta mezcla de teclas. Habitualmente, se puede obviar este tipo de errores si se siguen los métodos sintetizados para las teclas de comandos determinados por el sistema operativo que maneja.

Este nivel de igualdad es de bajo nivel. Los creadores de interfaces constantemente deben tratar de lograr esta altura en una interfaz de usuario. Algunas veces, la semejanza de nivel alto así mismo es codiciada. Por ejemplo, puede ser provechoso llevar a cabo las mismas instrucciones (como ilustrar, duplicar, etcétera) sobre todos los tipos de entes del sistema. No obstante, Grudin señala que la similitud total no siempre es viable o esperada. Puede ser sensato efectuar el borrado de un escritorio deslizando los entes a una papelera de reciclaje, pero sería fatigoso borrar el contenido en un procesador de contenidos de este modo.

Infelizmente, los principios de confianza del beneficiario e igualdad a veces son contrarios. Irrealmente, las aplicaciones con rasgos comunes convendrían utilizar siempre las mismas órdenes para acceder a estos rasgos. Pero, esto puede chocar con las costumbres del usuario cuando los sistemas se bosquejan para favorecer a un tipo de usuario especialmente, como los creadores gráficos. Estos usuarios logran tener que desenvolver sus propios modos de interacciones, terminología

| Principio | Descripción |
|---|---|
| **Familiaridad del usuario** | La interfaz debe utilizar términos y conceptos de la experiencia de las personas que más utilizan el sistema. |
| **Uniformidad** | Siempre que sea posible, la interfaz debe ser uniforma en el sentido de que las operaciones comparables se activen de la misma forma. |
| **Mínima sorpresa** | El comportamiento del sistema no debe provocar sorpresa a los usuarios. |
| **Recuperabilidad** | La interfaz debe incluir mecanismos para permitir a los usuarios recuperarse de los errores. |
| **Guía de usuario** | Cuando ocurran errores, la interfaz debe proporcionar retroalimentación significativa y características de ayuda sensible al contexto. |
| **Diversidad de usuarios** | La interfaz debe proporcionar características de interacción apropiadas para los diferentes tipos de usuarios del sistema. |

***Tabla 1.*** Principios de diseño de las interfaces de usuario.

y acuerdos de trabajo. Éstas pueden estar en disconformidad con los « modelos » de interacción que son adecuados para estudios más generales como los procesadores de contenidos.

El principio de mínima sorpresa es adecuado debido a que los individuos se impacientan excesivamente cuando el sistema procede de forma imprevista. Cuando se maneja un sistema, los usuarios edifican una guía mental de la forma en que trabaja dicho sistema. Si un hecho en algún contenido induce un tipo de canje particular, es prudente esperar la misma operación en un contenido diferente cause un cambio semejante. Si ocurre algo totalmente diferente, el usuario se asombra y enreda. Por lo tanto, los diseñadores de interfaces deben pretender afirmar que las labores semejantes tengan efectos confrontables.

Las sorpresas en las interfaces de usuario frecuentemente se deben al hecho de que en muchas interfaces constan varios modos de labor (por ejemplo, el modo panorámico y el modo impresión), y el resultado

de una instrucción es otra dependiendo del modo. Es muy significativo que, al bosquejar una interfaz, se incluya un guía visual que exponga al usuario el modo real.

El principio de recuperabilidad es primordial debido a que los usuarios irremediablemente cometen faltas cuando manejan un sistema. El bosquejo de la interfaz puede disminuir estas faltas (por ejemplo, las faltas de teclado se remedian si se manejan menús), pero las faltas nunca logran descartarse completamente. Por consecuente, se deben contener recursos que consientan a los usuarios recobrarse de sus faltas. Éstos pueden ser de tres tipos:

1. Confirmación de acciones destructivas. Si un usuario lleva a cabo una labor que es latentemente destructiva, el sistema debería solicitar que ratifique que esto es efectivamente lo que quiere antes de deshacer cualquier indagación.
2. Proporcionar un recurso para deshacer. La técnica deshacer restituye el sistema al momento previo antes de que sucediera la operación. Son ventajosos varias medidas de este medio debido a que los usuarios no siempre examinan seguidamente que han cometido una falta.
3. Generar puntos de control. La reproducción de espacios de inspección involucra grabar la etapa de un sistema en momentos habituales y admitir que el sistema se reponga desde el actual punto de inspección. De este modo, cuando se origina una falta, los usuarios consiguen volverse a una etapa previa y comenzar de nuevo. En la actualidad, muchos sistemas contienen la reproducción de puntos de comprobación para conocer los errores del sistema pero, paradójicamente, no admiten a los usuarios del sistema manipularlos para restablecerse de sus propias fallas.

Un principio a fin es el de asistencia al usuario. Las interfaces deben proveer ayuda al usuario o tipos de asistencia. Éstas se deben completar en el sistema y suministrar otros niveles de asistencia y orientación. Los niveles deben cambiar desde la información primordial para iniciarse con el sistema hasta una representación completa de las particularidades del sistema. Los sistemas de servicio se deben constituir de forma que cuando el usuario solicite apoyo no se sienta fastidiado con la información.

El principio de diversidad de usuarios determina que, para diversos sistemas interactivos, pueden constar otros tipos de usuarios. Unos son usuarios ocasionales y tienen interacción con el sistema frecuentemente, mientras que otros son usuarios eventuales que manipulan el sistema durante varias periodos de tiempo todos los días. Los usuarios ocasionales requieren interfaces que los dirijan, pero los usuarios eventuales requieren métodos resumidos de forma que consigan interactuar con el sistema tan pronto como sea posible. También, los usuarios logran tener dificultades de varios tipos y, si es viable, las interfaces deben poder ajustarse para hacer cada a esto. Consecuentemente, se lograría incluir recursos para exponer otros tamaños de contenido, sustituir el sonido con texto, permite cambiar el tamaño de los botones, etcétera. Esto muestra la generalidad de Diseño Universal (UD), un principio de diseño cuyo propósito es impedir descartar usuarios debido a alternativas de diseño involuntarias.

El principio de investigación de la variedad de usuarios logra estar en desafío con los demás principios de diseño de las interfaces, ya que unos usuarios logran elegir una interacción muy veloz sobre, por ejemplo, la similitud de la interfaz. De modo similar, el nivel de apoyo solicitado puede ser absolutamente diferente para diferentes usuarios, y logra ser imposible ampliar apoyos convenientes para todos los tipos de usuario. Consiguientemente, se debe conseguir convenios para hacer factibles las carencias de estos usuarios.

### 2.3.1. Asuntos de diseño

Antes de empezar el desarrollo de diseño de la interfaz de usuario, se presentan unos contenidos generales de diseño que tienen que ser apreciados por los diseñadores de interfaces de usuario. Esencialmente, el diseñador de una interfaz de usuario se traza dos asuntos clave:

1. 1. ¿Cómo debe interactuar el usuario con el sistema informático?
2. 2. ¿Cómo se debe mostrar la información del sistema informático al usuario?

Una interfaz de usuario razonable debe constituir la interacción del usuario y la publicación de la investigación. Esto puede ser dificultoso debido a que los diseñadores mantienen un término medio entre los modos más convenientes de interacción y exposición para la utilidad,

la alineación y práctica de los usuarios del sistema, y el equipo útil.

#### 2.3.1.1. Interacción del usuario

La interacción del usuario representa formular órdenes y datos agrupados al sistema informático. En los primeros ordenadores, la única forma de armar esto era a través de una interfaz de línea de órdenes, y se manejaba un lenguaje de propósito determinado para comunicarse con el ordenador. No obstante, este camino se orientó a los usuarios especializados y en la actualidad se han definido varias orientaciones que son más factibles de manejar. Shneiderman ha catalogado estos modos de interacción en cinco cualidades principales:

1. 1Manipulación directa. El usuario interactúa solo con los objetos de la pantalla. La manipulación directa habitualmente envuelve un mecanismo apuntador (un mouse, un lápiz óptico, un trackball o, en una pantalla táctil) que muestra el ente a manejar y la labor, la cual describe lo que se conviene hacer con ese objeto.
2. Selección de menús. El usuario elige un orden de una lista de eventos (un menú). Asimismo puede elegir otra cosa de la pantalla por manipulación directa, y la orden procede sobre él. En este posición, para quitar un registro, elegiría el icono del registro y posteriormente el comando de suprimido.
3. Rellenado de formularios. El usuario completa los campos de un formulario. Unos campos logran llevar menús adjuntos, y el formulario logra poseer «botones» de acción que, cuando se presionan, forjan a que se inicie una acción. Regularmente no manipulará esta perspectiva para realizar la interfaz de procedimientos como el suprimido de registros. Hacer esto involucraría introducir el alias del registro en el formulario y posteriormente «presionar» un botón de suprimir.
4. Lenguaje de comandos. El usuario escribe una orden exclusiva y los parámetros asociados para mostrar al sistema qué proceder. Para suprimir un registro, se probaría una orden de suprimido con el del registro como parámetro.
5. Lenguaje natural. El usuario expresa una orden en lenguaje natural. Habitualmente esto es un front-end para un lenguaje de órdenes; el lenguaje natural se examina y convierte a órdenes del

sistema. Para suprimir un registro, se probaría suprimir el registro.

Ciertamente, estos modos de interacción se logran combinar, manipulando diferentes modos en la propia aplicación. Por ejemplo, Microsoft Windows admite la manipulación directa de los iconos que interpretan a los registros y carpetas, la elección de órdenes establecidos en menús, y para órdenes como los de configuración, los usuarios deben completar un formulario de finalidad específica que se les modele.

En principio, es viable apartar el modo de interacción de las entidades inferiores que se manejan a través de la interfaz de usuario. Ésta fue el pie para el piloto de Seeheim sobre la realización de interfaces de usuario. En este piloto, se aparta la introducción de la información, la gestión del diálogo y la aplicación. En el contexto, éste es un piloto ideal más que un piloto experto, no obstante es indisputablemente posible poseer interfaces apartadas para los otros tipos de usuarios (usuarios ocasionales y usuarios habituales) que interactúan con el propio sistema inferior. Esto se ilustra en la Figura 54. La cual prueba una interfaz de lenguaje de órdenes y una interfaz gráfica para un sistema operacional como Linux.

Las interfaces de usuario fundamentadas en web se constituyen en el soporte facilitado por HTML o XHTML (el lenguaje de representación de páginas manejado por las páginas web) asociado con lenguajes como Java, los cuales consiguen relacionar programas con los elementos de una página. Debido a que regularmente estas interfaces fundamentadas en web son diseñadas por usuarios ocasionales, la generalidad de ellas manipula interfaces apoyadas en formularios. Es viable edificar interfaces de manipulación directa en web, pero ésta es un trabajo complicado de programación.

**2.3.1.2Presentación de la información**

Todos los sistemas interactivos tienen que suministrar una forma de mostrar la información a los usuarios. La exposición de la información logra ser puramente una representación inmediata de la información de ingreso (por ejemplo, contenido en un procesador de contenidos) o mostrar la información gráficamente. Una buena muestra de diseño es conservar apartado el software solicitado para la exposición de la información propia. Apartar el sistema de exposición de los antecedentes nos admite alternar la representación en la pantalla

del usuario sin realizar cambios el sistema de cálculo inferior. Esto se ilustra en la Figura 55.

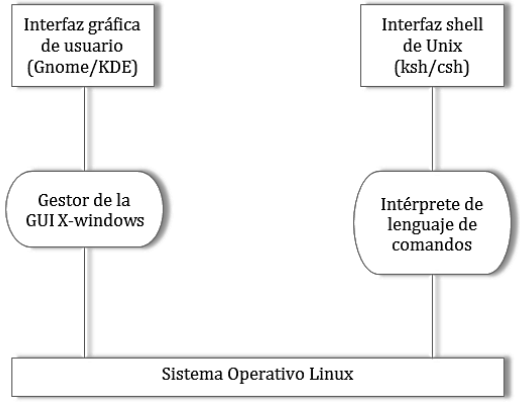



***Figura 54.*** Múltiples interfaces de usuario

La orientación MVC (Figura 56), el cual quedó utilizable por primera vez en Smalltalk, es una forma segura para consentir representaciones variadas de datos. Los usuarios logran interactuar con cada exposición manipulando un modo adecuado a ésta. Los datos a representar se encapsulan en un tipo de objetos. Cada tipo de objetos puede poseer asociados diversos objetos vista desigual donde cada vista es una representación de visualización desigual del modelo.

Cada vista posee un objeto controlador asociado que manipula los ingresos del usuario y la interacción de los equipos. Consecuentemente, un tipo que simboliza datos numéricamente logra poseer una vista que interpreta los datos como un histograma y una vista que muestre

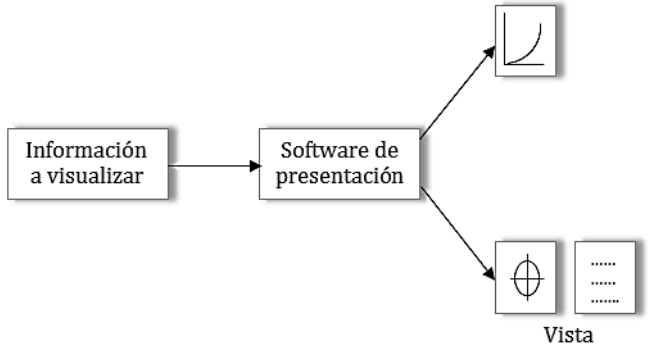



***Figura 55.*** Presentación de la información

los datos como una tabla. El modelo se consigue editar sustituyendo los valores en la tabla o prolongando o reduciendo las barras en el histograma.

Para hallar la mejor exposición de la información, se requiere conocer a los usuarios de la información y entender cómo dispondrán el sistema. Cuando se resuelve cómo mostrar la información, deben mantener presente las siguientes cuestiones:

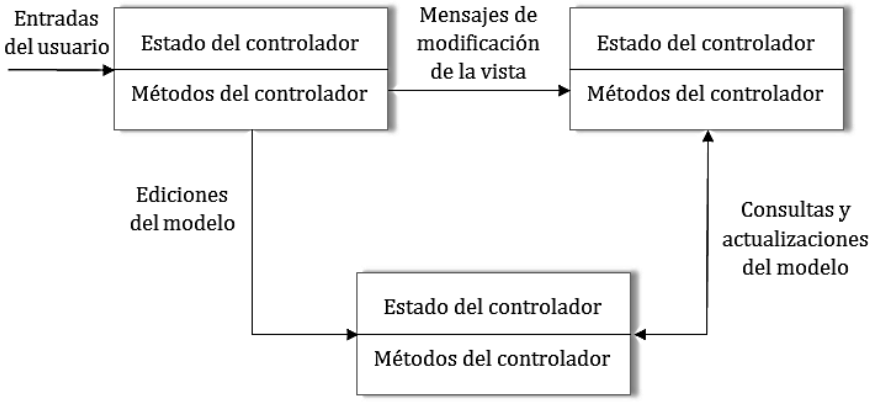



***Figura 56.*** El modelo MVC de interacción del usuario

1. ¿El usuario está atraído en información determina o en las dependencias entre los valores de los datos?
2. ¿Con qué frecuencia sustituyen los valores de la información? ¿Se mostrarán de forma rápida al usuario las variaciones en un valor?
3. ¿El usuario debe llevar a cabo alguna labor en réplica a las variaciones de la información?
4. ¿El usuario requiere interactuar con la información representada a través de una interfaz de manipulación directa?
5. ¿La información que se va a representar es textual o numérica? ¿Son significativos los valores referentes de los elementos de la información?

No se debe creer que por manejar gráficos se crean las vistas más atractivas. Los gráficos invaden un apreciable espacio en la pantalla (un asunto significativo en los dispositivos móviles) y alcanzan suficiente tiempo en descargarse si el usuario está haciendo con una conexión de

dirección telefónica pesada.

Dependiendo de la aplicación, la información que no se modifica durante una deliberación se puede mostrar tanto detallada como literalmente. La exposición textual invade menos área en la pantalla, pero no se consigue examinar de un vistazo. Debe diferenciarse la información que no se modifica de la información solícita manejando otros modos de exposición. Por ejemplo, podría mostrar toda la información invariable con una pauta de letra o matiz individual, o podría relacionar con una ilustración de «información invariable ».

Se debe manejar texto para mostrar la información cuando se solicita que ésta sea concreta y reemplace de forma respectivamente pesada. Si los datos permutan apresuradamente o si las relaciones entre los datos más que los valores correctos de los datos son significativos, se debe mostrar la información detalladamente.

Por ejemplo, piense un sistema que reconoce y sintetiza los valores de negocio periódicos de una compañía. La Figura 57 ilustra cómo mostrar la propia información como contenido o en forma detallada. Regularmente, los apoderados que aprenden las cantidades de ventas están más atraídos en las directrices o incoherencias que en los valores puntuales. La exposición detallada de esta información, como un histograma, hace que las cantidades originales en marzo y en mayo recalquen sobre las otras. La mencionada figura asimismo ilustra cómo la exposición textual invade menos área que la exposición detallada de la misma información.

En las salas de inspección o los tableros de órdenes como los del salpicadero de un vehículo, la información que se muestra interpreta el estado de algún otro sistema y está permutando consecutivamente. Las vistas digitales que invierten continuamente logran ser imprecisas e incómodas ya que los lectores no logran leer y asemejar la información antes de que se modifique. Consecuentemente, la información numérica que transforma activamente se constituye mejor de forma detallada manejando una representación analógica. Si es preciso, las vistas detalladas logran mejorar con una vista digital determina. En la Figura 58 se exponen otras formas de mostrar información numérica pronta.

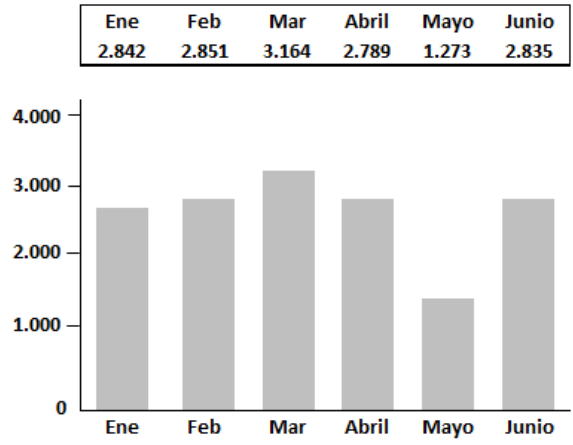



***Figura 57.*** Presentaciones alternativas de la información.

Cuando se tienen que mostrar amplias conjuntos de información, se logran manejar visualizaciones indeterminadas que enlacen los elementos de los datos afines. Esto puede manifestar relaciones que no son elementales en los datos sin formato. Se debe ser consecuente de los eventos de visualización, fundamentalmente cuando la interfaz de usuario del sistema debe caracterizar entes físicos. He aquí unos ejemplos de visualizaciones de datos:

1. La información atmosférica, coleccionada de diferentes fuentes, se prueba como un mapa atmosférico con isóbaras, faces atmosféricas, etcétera.
2. El curso de una red telefónica se demuestra gráficamente como

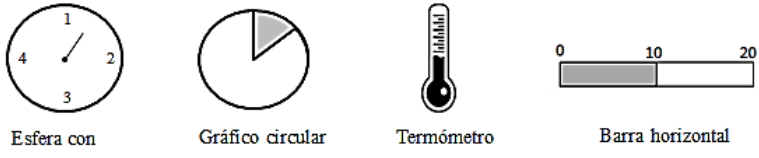



***Figura 58.*** Métodos de presentar información numérica que varía dinámicamente

un compuesto vinculado de nodos en un centro de dirección de la red.

3. El estado de una planta artificial se representa exponiendo las

amenazas y temples en un conjunto respectivo de almacenes y conducciones.

4. Un piloto de un elemento se evidencia y maneja en tres tamaños manejando un sistema de contexto virtual.
5. Un conjunto de páginas web se evidencia como un árbol amplificado.

Shneiderman, brinda una generosa visión habitual de las perspectivas para la visualización además de semejar las clases de visualización que se logran manipular. Estas contienen la visualización de datos manejando exposiciones en dos y tres espacios y en representación de árboles o redes. La totalidad de éstas se describen a la visualización de grandiosos montos de información tramitada en una computadora. Pero, el manejo más frecuente de la visualización en las interfaces de usuario es para interpretar alguna organización física, como la organización molecular de una nuevo compuesto, los enlaces en una red de telecomunicaciones, etcétera. Las exposiciones en tres dimensiones que logran manejar dispositivo de realidad virtual exclusivo son especialmente poderosos para originar visualizaciones. Un modo muy efectivo para interactuar con los datos es la manipulación directa de estas visualizaciones, tal como se observa en la figura 59.

También del modo de exposición de la información, se debe enfocar detenidamente en los colores manipulados en la interfaz. El color puede optimizar las interfaces de usuario favoreciendo a los usuarios a entender y manipular la diversidad. No obstante, es posible manejar el color de forma equivocada para establecer interfaces visualmente poco atrayentes y expuestas a faltas. Shneiderman da 14 modelos clave para el manejo efectivo del color en las interfaces de usuario. Las más significativas son:

1. Limitar el número de colores utilizados y ser conservador en la forma de utilizarlos. No conviene manipular más de cuatro o cinco colores desiguales en un formulario y no más de siete en una interfaz del sistema. Si se manipulan excesivos, o si son exageradamente vivos, la vista logra ser imprecisa. Unos usuarios logran pensar que las grandiosas cantidades de colores son incómodas y visualmente fatigosas. Igualmente es permitido el desconcierto en el usuario si los colores no se manejan de modo equivalente.

2. Utilizar un cambio de color para mostrar un cambio en el estado del sistema. Su vista reemplaza de color, debe representar

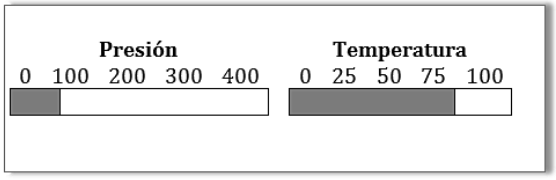



***Figura 59.*** Vista de Información gráfica que muestra valores relativos

que ha pasado un suceso significativo. Así, en una muestra del nivel de combustible, se podría manipular una variación de color para demostrar una bajada. Destacar el color es muy significativo en las vistas complicadas donde se exponen centenas de entidades diferentes.

3. Utilizar el código de colores para apoyar la tarea que los usuarios están tratando de llevar a cabo. Si los usuarios tienen que reconocer peticiones informales, se deben destacar estas peticiones; si todavía tienen que revelar semejanzas, se deben destacar éstas, manipulando un color distinto.
4. Utilizar el código de colores de una forma consciente y uniforme. Si una parte de un sistema prueba los mensajes de falla en rojo (por ejemplo), todas las demás partes deben exponer de igual forma. El rojo no se debe manejar para nada más. Si se hace, es viable que el usuario exponga la vista en rojo como un mensaje de falla.
5. Ser cuidadoso al utilizar pares de colores. Debido a la acción del ojo del hombre no logran orientar el rojo y el azul al mismo tiempo. La vista agotada es un resultado factible de una vista en rojo sobre azul. Diferentes combinaciones de colores logran ser además visualmente incómodas o dificultosas de leer.

En general, se debe manipular el color para la trabajo de sobresalir, pero no se debe relacionar significados con colores individuales. Alrededor del 10% de los hombres son daltónicos y logran malinterpretar el significado. Las impresiones humanas del color son desiguales, y existen acuerdos diferentes en otras carreras acerca del significado de

colores individuales. Los usuarios con preparaciones desiguales irresponsablemente pueden entender el mismo color de formas diferentes.

Al mismo tiempo de mostrar la información de la aplicación, los sistemas de igual forma se notifican con los usuarios a través de mensajes que facilitan información sobre las fallas y el estado del sistema. La primera acción de un usuario de un sistema software logra ser cuando el sistema muestra un mensaje de falla. Los usuarios inhábiles logran emprender su labor, cometer una falla primordial y de forma rápida tienen que entender el mensaje de falla resultante. Esto logra ser bastante dificultoso para los ingenieros de software especializados. Es frecuentemente inalcanzable para los usuarios principiantes u ocasionales del sistema.

Se debe evitar la alineación y práctica de los usuarios cuando se diseñan mensajes de falla. Por ejemplo, imagine que un usuario del sistema es una asistente en una sala de cuidados intensivos de una clínica. La información del paciente se lleva a cabo mediante un sistema informático. Para ver el curso real del paciente, la asistente elige «mostrar» de un menú e inserta el nombre del paciente en una casilla, como se muestra en la Figura 60.

En este caso, vamos a suponer que la asistente ha escrito erróneamente el nombre del paciente y ha presionado «MacDonald» en vez de «McDonald». El sistema crea un mensaje de falla.

Los mensajes de falla eternamente deben ser serios, breves, iguales y provechosos. No deben ser ofensivos ni tener sonidos agrupados u otro tipo de sonidos que logran turbar al usuario. En la dimensión de lo viable, el mensaje debe indicar cómo se podría corregir el error. El mensaje de error debe relacionarse a un sistema de apoyo en línea visible al contenido.

La Figura 61 muestra ejemplos de mensajes de errores perfectos e incorrectamente bosquejados. El mensaje de la izquierda está mal bosquejado. Es perjudicial, no se ajusta a las prácticas y al grado de práctica del usuario, y no mantiene en cuenta la información del contenido. No indica cómo se podría modificar la realidad. Maneja métodos

| Factor | Descripción |
|---|---|
| **Contexto** | Donde sea posible, los mensajes generados por el sistema deben reflejar el contexto actual del usuario. En el posible, el sistema debe ser consciente de lo que está haciendo el usuario y generar mensajes relacionados con su actividad actual. |
| **Experiencia** | Al aumentar la familiaridad de los usuarios con el sistema, también se aumenta su molestia por los mensajes largos y "significativos". Sin embargo, los principiantes tienen dificultades en comprender los mensajes cortos y concisos del problema. Se deben proporcionar ambos tipos de mensajes y permitir al usuario controlar la concisión de los mensajes. |
| **Nivel de habilidad** | Los mensajes se deben adaptar a las habilidades del usuario, así como a su experiencia. Los mensajes para las diferentes clases de usuario se pueden expresar de diferentes clases de usuario se pueden expresar de diferentes formas dependiendo de la terminología que el lector utilice. |
| **Estilo** | Los mensajes deben ser positivos en vez de negativos. Deben estar escritos en modo activo y no en pasivo. No deben ser insultantes a tratar de ser graciosos. |
| **Cultura** | En la medida de lo posible, el diseñador de mensajes debe estar familiarizado con la cultura del país donde se vende el sistema. Existen distintas diferencias culturales entre Europa, Asia y América. Un mensaje adecuado en una cultura podría no aceptarse en otra. |

***Tabla 2.*** Factores que implican un mensaje de error.

concretos del sistema (identificador de paciente) en vez de un lenguaje encaminado al usuario. El mensaje de la derecha es superior. Es real, lo que da a pensar que la dificultad es del sistema y no del usuario. Identifica el problema en cláusulas entendibles para la asistente y brin-

da una forma cómoda para corregir el error tocando un solo botón. El sistema de apoyo está aprovechable si se precisa.

### 2.3.2. El proceso de diseño de la interfaz de usuario

El diseño de la interfaz de usuario (UI) es una fase donde los usuarios interactúan con los diseñadores y prototipos de la interfaz para resolver los tipos, ordenación, aspecto y labor de la interfaz de usuario del sistema. Ocasionalmente, se edifica el modelo de la interfaz por separado al mismo tiempo con otras actividades de la ingeniería del software. Más usualmente, en específico cuando se maneja un progreso iterativo, el diseño de la interfaz de usuario se lleva de manera incremental acorde se desarrolla el software. En ambos asuntos, no obstante, antes de que comience la programación, debe haber perfeccionado e, idealmente, comprobado unos diseños en papel.

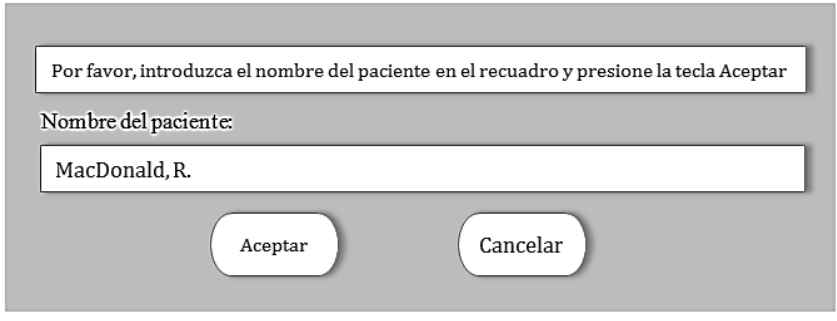



***Figura 60.*** Un recuadro de entrada de texto utilizado por una asistente

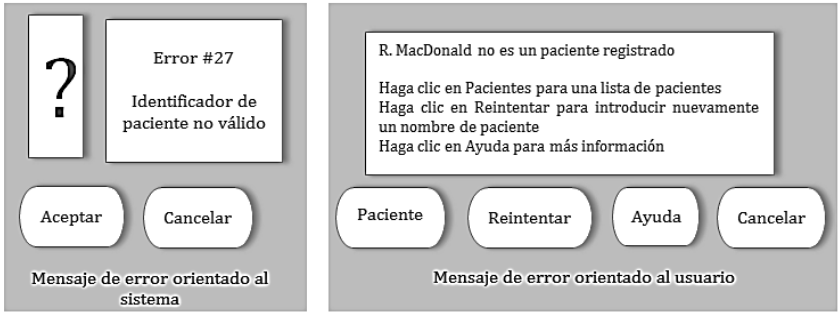



***Figura 61.*** Mensajes de error al sistema y al usuario

El proceso de diseño general de la UI. Existen tres acciones fundamentales en este proceso:

1. Análisis del usuario. En el análisis del usuario, se comprende las tareas que éste realiza, en su entorno de trabajo, junto con otros sistemas que utiliza, como se desenvuelve con el resto de las personas con las que labora, etc. Para productos que poseen usuarios de todo tipo, se pretende desarrollar la compresión a través de grupos de discusión, pruebas con diferentes usuarios y ejercicios similares.
2. Prototipado del sistema. El diseño y progreso de la interfaz de usuario es un desarrollo iterativo. No obstante los usuarios logran conversar de las habilidades que requieren de una interfaz, es muy dificultoso para ellos ser concretos hasta que ven algo evidente. Consecuentemente, se han desarrollado modelos del sistema y mostrar a los usuarios, quienes logran entonces guiar el progreso de la interfaz.
3. Evaluación de la interfaz. Sin embargo obviamente se tendrá que hablar con los usuarios durante el transcurso de prototipado, además se debe poseer una acción de valoración más reglamentaria donde se coleccione información sobre las prácticas existentes de los usuarios con la interfaz.

Este aparato se concentra en la investigación del usuario y en la valoración de la interfaz, con sólo una breve representación de los procesos específicos de prototipado de interfaces de usuario.

La elaboración de calendarios del diseño de la Ul adentro del proceso del software obedece, hasta cierto punto, de nuevas acciones. Como se vio en el Capítulo 7, se consigue manipular la edificación de prototipos como parte del transcurso de la ingeniería de requerimientos y. en este asunto, tiene sentido emprender el proceso de diseño de la UI en esta fase. En los métodos iterativos el diseño de la Ul se completa con el proceso del software. Como el software propio, se puede poseer que refactorizar y retocar la UI durante el proceso.

### 2.3.3. Análisis del usuario.

Una acción crítica del diseño de la UI es el estudio de las acciones del usuario que deben ser aguantadas por el sistema informático. Si no se comprende lo que los usuarios desean hacer con el sistema, no se lo-

grará producir un diseño real y efectivo de la interfaz de usuario. Para ampliar este conocimiento, puede manipular procesos como el estudio de trabajos, estudios técnicos, entrevistas de usuarios e investigaciones o usualmente, una mezcla de ellas.

Un desafío para los ingenieros envueltos en el análisis de usuarios es hallar una forma de representar los análisis de manera que notifiquen la naturaleza de las labores a los distintos diseñadores y a los usuarios propios. Signos como los diagramas de secuencia de UML logran representar las interacciones del usuario y son excelentes para informarse con los ingenieros de software. No obstante, distintos usuarios logran pensar que estos diagramas son excesivamente expertos y no ansiarán entenderlos. Debido a que es muy significativo envolver a los usuarios en el transcurso de diseño, habitualmente tendrá que desarrollar escenarios en lenguaje natural para representar las acciones del usuario.

En un ejemplo de un contexto en lenguaje natural que se lograría haber especificado durante el trascurso de descripción y bosquejo del sistema LIBSYS. Describe una situación en la que no existe el LIBSYS y una estudiante necesita obtener información de otra biblioteca. De este escenario, el diseñador puede ver varios requerimientos:

Jane es una alumna de estudios creyentes que está arreglando un escrito sobre la edificación india y cómo ésta ha estado involucrada por los hábitos religiosos. Para ayudarle a comprender esto, le gustaría entrar a imágenes de detalles de edificaciones notables, pero no puede hallar nada en su librería local.

Se aproxima al bibliotecario para expresarle sus carencias y éste le propone técnicas de búsqueda que lograría manipular. Además le propone librerías en Nueva Delhi y Londres que lograrían poseer este material, por lo que ingresan a los catálogos de la librería e investigan manipulando estos métodos. Hallan una fuente de material y elaboran un requerimiento de fotocopias de los dibujos con los detalles arquitectónicos, para que se las manden directo a Jane.

1. Los usuarios lograrían no saber los métodos de investigación adecuados. Pueden precisar asumir acceso a formas de favorecerles para preferir métodos de investigación.
2. Los usuarios han de poder elegir colecciones para investigar.
3. Los usuarios requieren poder sobrellevar a cabo investigacio-

nes y requerir copias de material apreciable.

No se debe esperar que el estudio del usuario cree requerimientos de interfaces de usuario muy determinados. Regularmente, el estudio apoyo a la comprensión las carencias e inquietudes de los usuarios del sistema. Acorde se sabe mejor cómo trabajan y cuáles son sus inquietudes y limitaciones, éstas logran mantener en cuenta en el diseño. Esto significa que es más posible que los diseños preliminares sean admitidos por los usuarios y, luego, convencerlos para envolverlos en el transcurso de beneficio del diseño.

#### 2.3.3.1. Técnicas de análisis

Como se indicó en la sección anterior, constan de tres métodos base de análisis del usuario: análisis de tareas, conferencias y preguntas, y etnografía. El análisis de tareas y las conferencias se concentran en la persona y en su labor, mientras que la etnografía ayuda a una apariencia más habitual y encuentra cómo interactúan las personas, cómo constituyen su ambiente de labor y cómo ayudan para solucionar dificultades.

Existen algunas clases de análisis de tareas, pero la más frecuentemente manipulada es el Análisis de Tareas Jerárquico (HTA). El HTA fue difundido en un principio para apoyar a redactar manuales de usuario, pero además se logra manipular para reconocer lo que crean los usuarios para lograr algún objetivo. En el HTA, una tarea de valioso nivel se secciona en subtareas, y se identifican procedimientos que detallan lo que ocurrirían en una ambiente determinado.

Principiando con un objetivo del usuario, se bosqueja una categoría que prueba qué se tiene que crear para lograr ese objetivo. En la notación del HTA, una línea debajo de la caja regularmente muestra que no se deshará en subtareas más minuciosas.

La ventaja del HTA sobre los contextos en lenguaje natural es que exige a pensar cada una de las labores y concluir si éstas se convienen destruir. Con los contextos en lenguaje natural, es posible excluir trabajos significativos. Los contextos asimismo se hacen extensos y fastidiosos de leer si se quiere agregarles varios datos.

El inconveniente con esta perspectiva para representar las tareas del usuario es que es más adecuado para trabajos que son técnicas secuenciales. La notación se hace dificultosa cuando se pretende mo-

delar trabajos que envuelven actividades enlazadas o compatibles que soportan varias subtareas. Además, el HTA no registra por qué los trabajos se crean de una forma en particular o limitan las técnicas del usuario.

Habitualmente, se reúne información para el HTA a través de la investigación y de conferencias con los usuarios. En este transcurso de conferencias, se puede reunir parte de esta información esencial y registrarla al lado de los análisis de tareas. Cuando se ejecutan conferencias para revelar lo que los usuarios crean verdaderamente, se han de bosquejar las conferencias de forma que los usuarios logren facilitar información que ellos asuman que es significativo.

Las conferencias, ciertamente, no son sólo una forma de conseguir información para el análisis de trabajos: son una habilidad general para lograr información. Se logran perfeccionar las entrevistas propias con entrevistas en conjunto o conjuntos de disputa. La ventaja de manipular grupos de conversación es que los usuarios se inducen entre sí para ofrecer información y logran finalizar discutiendo de distintas formas que han desarrollado de manejar los sistemas.

El estudio de trabajos se concentra en cómo trabajan los individuos, pero, ciertamente, la totalidad del trabajo es verdaderamente cooperativo. Las personas trabajan agrupados para lograr un objetivo, y los usuarios hallan dificultoso discutir cómo se lleva a cabo esta contribución. Luego, la investigación directa de cómo trabajan y manejan los sistemas informáticos los usuarios es una significativa habilidad adicional de análisis del usuario.

Como ejemplo de cómo logra influir la etnografía en el diseño de las interfaces de usuario, es un segmento de un informe de un estudio etnográfico de los inspectores del comercio aéreo. Vivíamos atraídos en el diseño de la interfaz para un sistema de inspección del tráfico aéreo (ATC) más mecanizado y asimilamos dos sucesos significativos de estas investigaciones:

1. Los consoladores no lograban ver unos de los vuelos de una parte. Consecuentemente, correspondemos impedir manipular ventanas que utilicen barras de deslizamiento donde los vuelos concluyan por la parte principal o secundaria de las ventanas.
2. La interfaz debe poseer una forma de notificar a los inspectores cuántos vuelos hay en las partes contiguos de forma que logren

planear su carga de labor.

Evidenciar las partes contiguos era una gestión automática del controlador y es muy posible que no la hubieran sugerido en las cuestiones sobre el transcurso del ATC. Revelamos estos significativos avisos sólo a través de la observación directa.

### 2.3.4. Prototipado de la interfaz de usuario.

Debido al ambiente eficiente de las interfaces de usuario, las representaciones textuales y los esquemas no son convenientes para enunciar los requerimientos de éstas. El prototipado progresivo o experimental con la importancia de los usuarios finales es la única forma experiencia de bosquejar y desplegar interfaces gráficas de usuario para métodos software. Envolver al usuario en el transcurso de bosquejo y proceso es un aspecto primordial del bosquejo central en él, un razonamiento de bosquejo para sistemas recíprocos.

La intención del prototipado es ayudar a los usuarios lograr una práctica directa con la interfaz. La totalidad de nosotros halla dificultoso pensar de forma neutra sobre una interfaz de usuario y exponer puntualmente qué anhelamos. No obstante, cuando se nos muestran modelos, es posible igualar las particularidades que nos agradan y las que no. Irrealmente, cuando se está edificando el prototipo de una interfaz de usuario, se debe acoger un transcurso de prototipado en dos cursos:

1. Al principio del transcurso, hay que elaborar prototipos en documento modelos de los bosquejos de las pantallas y exponer a los usuarios finales.
2. Entonces, se afina el bosquejo y se elaboran prototipos automatizados cada vez más rebuscados, y se colocan a disposición de los usuarios para ejecutar experimentos y simulacro de actividades.

Para la realización de prototipos. No se precisa realizar ningún software factible y los bosquejos no poseen por qué crear acorde a tipos expertos. Se logran crear exposiciones en un documento de las pantallas del sistema con las que interactúan los usuarios y se logra bosquejar un conjunto de escenarios que relaten cómo se lograría manipular el sistema. Acorde se explica un escenario, hay que bosquejar la información que se expondrá y las opciones utilizables para los usuarios.

Se debe hacer entonces a través de estos escenarios con los usuarios para representar el modo en que se manipulará el sistema. Es una

forma segura de ver las reacciones preliminares de los usuarios a un bosquejo de la interfaz, la investigación que requieren del sistema y cómo interactuarán regularmente con el sistema.

Se logran llevar a cabo ensayos de prototipado adicionales manejando tanto una orientación progresiva como una orientación recusable. Viven tres orientaciones que logran manipularse para el prototipado de interfaces de usuario:

1. Enfoque dirigido por secuencias de comandos. Si simplemente se requiere experimentar ideas con los usuarios, se logra manipular una orientación dirigida por series de órdenes, como el que hallará en Macromedia Director. En esta orientación, se crean pantallas con partes visuales, como botones y menús, y se agrega una serie de órdenes con estas partes. Cuando el usuario interactúa con estas pantallas, se hace la serie de órdenes y se muestra la sucesiva pantalla, que les muestra los efectos de sus operaciones. No hay comprometida ninguna función lógica.
2. Lenguajes de programación visuales. Los lenguajes de programación visuales, como Visual Basic, agregan un poderoso ambiente de proceso, permiten a una gran diversidad de entes reutilizables y a un sistema de proceso de interfaces de usuario que consiente crear interfaces de forma instantánea, con elementos y series de órdenes agrupados con los entes de la interfaz. Se narran los sistemas de proceso visuales en el Capítulo 7.
3. Prototipado basado en Internet. Estos procedimientos, asentadas en navegadores web y en lenguajes como Java, brindan una interfaz de usuario creada. Se aumenta funcionalidad relacionando fragmentos de programación Java con la investigación a visualizar. Estos fragmentos se hacen involuntariamente cuando se carga la página en el navegador. Esta orientación es un modo instantáneo de ampliar prototipos de interfaces de usuario, pero quedan prohibiciones esenciales aplicadas por el navegador y por el piloto de seguridad de Java.

Comprensiblemente, el prototipado está muy conexo con la valoración de la interfaz. Es imposible que las valoraciones serias estén económicas para los principales prototipos, ya que lo que se está pretendiendo lograr en este curso es una «valoración formativa» en la que se ave-

riguan modos de óptima la interfaz. Acorde el prototipo se hace más exacto, pueden manipular procesos de valoración técnicas, como se conocerá en la sucesiva sección.

### 2.3.5. Evaluación de la interfaz.

La valoración de la interfaz es el trascurso de valorar la forma en que se maneja una interfaz y comprueba que cumple las imposiciones del usuario. Consecuentemente, debe ser fragmento del trascurso de comprobación y confirmación de los sistemas software. Neilsen, en su libro sobre ingeniería de usabilidad, contiene un buen apartado sobre este tema.

De forma excelente, una valoración se debe llevar a cabo hacia una descripción de la usabilidad fundamentada en propiedades de la usabilidad. Las métricas de estas propiedades de usabilidad se logran suponer. Por ejemplo, en una métrica de enseñanza, se lograría exponer que a un operador acostumbrado con las labores realizadas le debe ser viable manipular el 80% de la funcionalidad del sistema posteriormente de tres horas de formación. Pero, es más frecuente detallar la usabilidad de forma cuantitativa en vez de manipular métricas. Consecuentemente, el diseñador tiene que manipular su juicio y práctica en la valoración de las interfaces.

La valoración metodología del bosquejo de la interfaz de usuario logra ser un trascurso costoso que involucra a científicos conocedores y diseñadores gráficos. Es viable que se tenga que bosquejar y ejecutar un número estadísticamente significativo de ensayos con los usuarios propios. Se puede precisar el uso de estancias edificados fundamentalmente con equipos de inspección. Una valoración de la interfaz de usuario de esta pauta es económicamente poco objetiva para sistemas perfeccionados por pequeñas organizaciones con ahorros finitos.

Constan varios métodos menos trabajosos y fáciles en la valoración de interfaces que logran comprobar faltas determinadas en el diseño de interfaces:

1. Preguntas que coleccionan información de lo que expresan los usuarios de la interfaz.
2. La indagación de los usuarios cuando elaboran con el sistema y « imaginan en voz alta» de cómo tratan de manejar el sistema para llevar a cabo una tarea.

3. Momentáneas » de vídeos del uso tradicional del sistema.
4. La inserción de código en el software que compila información de los recursos más manipulados y de las fallas más comunes.

Valorar una interfaz. Las consultas deben ser concretas más que generales. No es de provecho hacer preguntas como «Por favor, haga comentarios sobre la usabilidad de la interfaz», puesto que las respuestas posiblemente modificarán tanto que no se revelará ninguna tendencia habitual. En su lugar, las preguntas determinadas como «Por favor, muestre el valor en un nivel de 1 a 5 de cuál es el conocimiento de los mensajes de falla. Un valor 1 representa muy clara y 5 representa incomprensible» son principales. Son más factibles de reconocer y es más posible que suministren información útil para perfeccionar la interfaz.

**Ejercicios**

1. Describir la diferencia entre un objeto y una clase, utilice 2 ejemplos de cada una.
2. Mediante un cuadro sinoptico describa la importancia de implementar el diseño orientado a objetos en el desarrollo de un sistema.
3. Describir que son los diagramas UML y realizar un ejemplo de diagramas de clases.
4. Diseñe un diagrama de secuencia de un proceso denominado obtener factura, de un sistema de facturación.
5. Diseñar un caso de uso del proceso cliente en el ejercicio anterior.
6. Por qué los sistemas de tiempo real tienen que implementarse normalmente utilizando procesos concurrentes. Uilice ejemplos si es posible.
7. Por qué en el desarrollo del software, una aproximación orientada a objetos no puede ser adecuada para sistemas de tiempo real.
8. Qué técnicas de diseño de sistemas de tiempo real estudiadas en neste capitulo son ijmportantes para larecogida de datos de la estación metereológica.
9. En que escenarios no es conveniente o viable facilitar una interfaz de usuario uniforme.
10. ¿Qué elementos deben tenerse en cuenta cuando se bosqueja

una interfaz asentada en menús para sistemas él es modelo de un cajero automático (ATM) de un banco? Escriba una observación crítica sobre la interfaz de un ATM que maneje.

11. Señale las ventajas de la visualización clara de la información y proponga cuatro aplicaciones donde sea más adecuado manipular ventanas gráficas en vez de ventanas digitales de información numérica.

# Resumen

- El diseño orientado a objetos permite que los elementos del diseño simbolicen objetos con su estado privado y en vez de funciones con operaciones.
- La implementación de los objetos se la realiza de forma secuencial, es decir cuando un objeto se encuentra en estado pasivo, cuyo estado se puede modificar mediante su interfaz; o cuando se encuentra en estado activo el cual puede cambiar sin participación externa.
- El proceso del diseño orientado a objetos consiste en: Diseño arquitectónico del sistemas, detallar los objetos involucrados en el sistema, documentar las interfaces de los objetos, usar un modelo para detallar el diseño
- Esto permite al diseñador conocer si los usuarios con cierta pauta de preparaciones poseen problemas con la interfaz. Los informes se logran manipular aun antes de que esté disponible el sistema factible si se edifican y evalúan modelos en papel de la interfaz.
- La valoración basada en la indagación simplemente comprende observar cómo manipulan el sistema los usuarios, ver los recursos que manipulan, las fallas cometidas, etcétera. Esto se logra complementar con reuniones de pensar en voz alta, en las cuales los usuarios hablan sobre lo que tratan de hacer, qué piensan del sistema y cómo tratan de manipularlo para llevar a cabo sus objetivos.
- En conclusión, es fácil suministrar a los usuarios un orden que logren manipular para remitir mensajes al diseñador de la herramienta. Esto hace que los usuarios crean que sus opiniones son tenidas en cuenta. Así, el diseñador de la interfaz y otros ingenieros consiguen obtener una ágil retroalimentación de los problemas particulares.

# Metodologías para el diseño de sistemas

## Descripción del contenido

En este capítulo: metodologías para el diseño de sistemas, se desarrolan diferentes subtemas que se detallan a continuación:

El objetivo general del punto 3.1. Desarrollo rápido de software, es detallar algunos enfoques para el software pensados para una entrega rápida del mismo. A continuación se lista los temas de importancia a tratar:

- Métodos ágiles
- Programación extrema
- Desarrollo rápido de aplicaciones
- Prototipado del software

El objetivo general del punto 3.2. Reutilización del software, conocer los beneficios y las dificultades de reutilizar software y formas de efectuarla. A continuación se lista los temas de importancia a tratar:

- El campo de la reutilización
- Patrones de diseño
- Reutilización basada en generadores
- Marcos de trabajo de aplicaciones
- Reutilización de sistemas de aplicaciones

### 3.1 Desarrollo rápido de software

En la actualidad, las empresas actúan en un ámbito global que se innova muy rápidamente, por tal motivo tiene que adaptarse a las nuevas oportunidades, plazas, la aparición de nuevos productos, constantes cambios en la economía y prestación de servicios por parte del competidor. El software es esencial en todas las tareas y operaciones del negocio, por lo cual es de suma importancia que el software se lo desarrolle lo más rápido posible, para ganar ventaja sobre la competencia y así aprovechar todas las oportunidades que se nos presente, por lo que muchas compañías están dispuestas olvidar la calidad del software y el acuerdo sobre los requerimientos para obtener una entrega de los requisitos más indispensables, corriendo el riesgo de futuras molestias.

Debido al entorno cambiante en el que laboran las empresas, es prácticamente imposible obtener requerimientos estables. Los requerimientos que se planteen de manera inicial cambiaran inevitablemente, ya que el cliente no puede predecir de qué manera el sistema afectara la forma de trabajar en su negocio, ni la interacción con los demás sistemas y cuales procesos se deberán automatizar. Solo quedan claros los requerimientos reales cuando el cliente tiene una interacción con el sistema.

Las metodologías basadas en la especificación de requerimientos, diseño, desarrollo y testing del sistema; no pertenecen a este grupo del desarrollo ágil de software. En el momento que surge un problema o un nuevo requerimiento en alguna de estas etapas, se tiene que volver a considerar estos nuevos aspectos, provocando que se alarguen algunas de las fases en el desarrollo del software; usualmente esto sucede en las metodologías tradicionales como la de cascada y la basada en la especificación, como consecuencia, el software final es entregado con mucha demora con respecto al tiempo limitante.

En un ambiente de negocios muy fluido y en constante cambio, esto se presenta como un gran problema. Puesto que un software que puede haber sido solicitado en un principio, se vuelva obsoleto para cuando ya esté terminado. Con el fin de solucionar estos inconvenientes, fueron concebidas las metodologías agiles de desarrollo de software agiles, las mismas que son muy usadas en la actualidad.

Para resolver el problema de proporcionar el producto al cliente lo

más rápido posible; el desarrollo ágil se caracteriza por tener procesos iterativos, que cumplen todas las etapas del software, pero por intervalos, llamados incrementos, es decir, que cada de uno de estos lleva nuevas funcionalidades del software. Existe diversos tipos de desarrollo ágil pero todos estos comparten sus características:

1. Tanto la fase de especificación, diseño e implantación coinciden en este tipo de desarrollo. No hay una especificación minuciosa, ni tampoco una documentación del diseño, comúnmente es generada automáticamente por el entorno de desarrollo en el cual se haya implementado. En los requerimientos el usuario solo determina los requisitos más críticos para el sistema.
2. Todos estos sistemas son desarrollo en una serie de incrementos los cuales son siempre una mejora del anterior. También se caracteriza por mantener al usuario final y a los stakeholders involucrados constantemente en las etapas de especificación y evaluación de cada incremento; dado que requiere de su participación para plantear nuevos requerimientos y proponer cambios en el software que pueden ser implementado en un nuevo incremento.
3. Frecuentemente se desarrollan interfaces de usuario, empleando IDE que permitan diseñar rápidamente un sistema interactivo. Estos sistemas pueden generar una interfaz basada en web o para plataformas específicas.

A continuación se presenta las ventajas más importantes de acoger un enfoque incremental.

1. Entrega acelerada de los servicios del cliente. En los primeros incrementos se proporciona las funcionalidades que tenga un alto rango de importancia para que el cliente utilice el sistema desde inicio. Lo cual es muy beneficioso tanto para el desarrollador, como para el cliente, debido a que este último puede ver sus requerimientos a través de la experiencia de la interacción y así especificar cambios que se incluirán en un nuevo incremento.
2. Compromiso del cliente con el sistema. El equipo desarrollador debe ayudar a que el usuario final se integre y comprometa en el proceso de desarrollo, ya su opinión, permitirá construir un software que cumpla más con sus requerimientos.

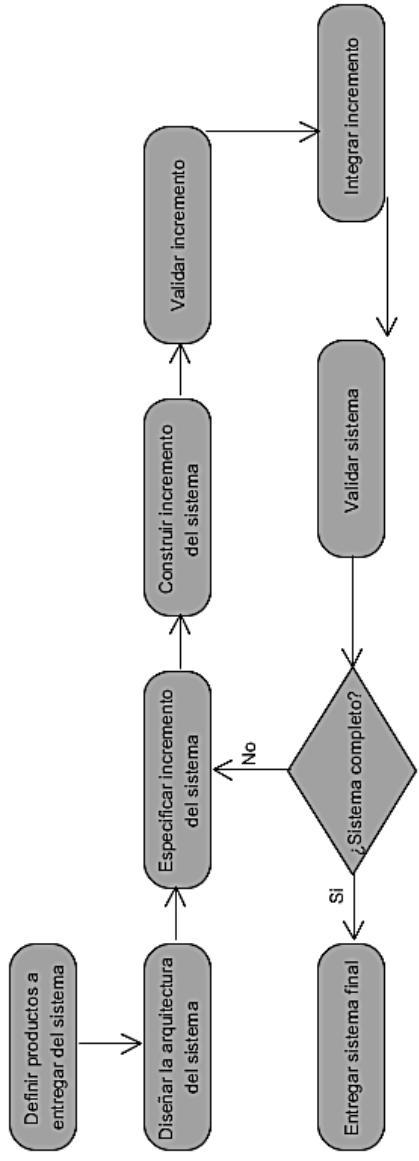



**Figura 62.** Un proceso de desarrollo iterativo

En la Figura 62 se ilustra una técnica para el desarrollo ágil e incremental. Se puede contemplar a simple vista que en las fases iniciales de este proceso se enfocan en el diseño arquitectónico; puesto que, si no se la considera desde un principio, provocara que el sistema sea inconstante y se degrade cada vez que se entregue un nuevo incremento

Debido a que no muy a menudo hallamos una solución que abarque todo el problema desde un principio, el desarrollo incremental es un enfoque muy favorable que refleja la forma común con la que se resuelven problemas en la vida cotidiana, es decir llegar a la solución a partir de pequeños pasos, y regresando al cometer algún error.

A pesar de ser una gran solución, este enfoque llega a ser un gran problema, en empresas donde sus procedimientos son muy inflexibles y sobre todo en aquellas donde se subcontrata agentes externos. Los problemas más comunes en el desarrollo iterativo y la entrega incremental son:

1. Problemas de administración. En este tipo de desarrollo existe constante cambio, por lo cual no es rentable elaborar tanta documentación, Además a los administradores del proyecto les resulta complicado encontrar personal con las capacidades aptas para este tipo de desarrollo incremental, en ciertos casos requiere utilizar tecnologías nuevas.
2. Problemas contractuales. Este modelo se plantea y concibe bajo la especificación de requisitos del sistema entre el cliente y un desarrollador de software. En caso que no existiera tal documento, difícilmente se puede llegar a firmar un contrato, debido a que los desarrolladores no aceptarían un contrato fijo, ya que existirán constantes cambios de los requerimientos por parte de los clientes, además de no sentirse satisfechos con un contrato en el que solo se pague por el proyecto, y no por el trabajo del desarrollador, generando una pérdida de tiempo y dinero.
3. Problemas de validación. En estos procesos basados en la especificación, la validación y verificación nos ayuda a garantizar la precisión del sistema en construcción, de tal manera disminuir problemas y satisfacer los requerimientos iniciales, esto nos ayuda a que se puedan lograr tareas en paralelo y minimizar la documentación, puesto que no se realiza independientemente,

sino más bien en conjunto.

4. Problemas de mantenimiento. Los constantes cambios en el software provocan inestabilidad e inconsistencia en el sistema, provocando que ni si quiera los desarrolladores originales llegan a comprender su propio software. Se recomienda modificar el código fuente pero sin cambiar su comportamiento permitiendo en cada incremento mejorar la estructura del software. Esto se indica en la sección 3.1.2, donde se trata la programación extrema y sobre los entornos de desarrollo rápido de aplicación (RAD).

Estos sistemas de desarrollo resultan ser muy útiles en sistemas muy grandes y en sistemas de embebido donde el software depende críticamente del hardware resultan ser obsoletas, otros casos en los que se nos presenta este tipo de problema son cuando los equipos se encuentra en varios sitios de la empresa o inclusive en diferentes ciudades provocando que se comprometa la seguridad de los datos de la empresa al no tomarse en cuenta algún requerimiento importante

Para reducir los problemas y beneficiarse con este desarrollo incremental, puesto que al igual que cualquier modelo desarrollo tradicional los requerimientos sufren constantes cambios; se puede optar por crear un proceso hibrido en el cual se pueda hacer pruebas tanto de diseño y de requerimiento a partir de prototipos del sistemas; ya que al tener un prototipo el usuario fácilmente puede realizar intervenir en la partes de prueba del sistema y así en un posterior incremento corregir errores e incluir nuevos requerimientos.

El prototipado se lo puede definir de dos maneras; primero como modelo de desarrollo experimental e interactivo que ayuda de muchas formas posibles principalmente que el cliente y los desarrolladores entiendan como se debe implementar el sistema. A la otra definición se le da como un pequeño incremento del prototipo anterior. En la Figura 63 se demuestran las diferencias entre sus objetivos tanto del prototipado incremental y la del desarrollo:

1. El modelo incremental se caracteriza por que en sus incrementos se desarrolla las funcionalidades más importantes para el negocio y que se hayan entendido con claridad.
2. El prototipado incremental tiene como objetivo empezar por los requerimientos más complicados de los cuales no se haya podi-

do comprender su funcionamiento. En su mayor parte nunca se necesita prototipos sencillos

Estos dos enfoques se diferencian también en la calidad que se le dé a los sistemas, ya que el prototipo incremental no se necesita que tenga un buen acabado de rendimiento y de fidelidad porque tienen un periodo de mi vida corto por sus constantes cambios en cada incremento. En cuanto a lo que se refiere al desarrollo incremental, estos desde un inicio tienen que ser construidos de una forma en la cual exista eficiencia y fiabilidad, debido a que estos sistemas se les deberán dar mantenimiento en un futuro, respetando los estándares de un software normal

### 3.1.1 Métodos ágiles

Durante la época en 80 y 90, se tenía la idea de que la mejor forma de desarrollar software era a partir de una buena planificación y la utilización de herramientas case en las etapas de análisis y diseño. Todas estas ideas fueron planteada por ingenieros de software que habían construidos sistemas de software con larga vida.

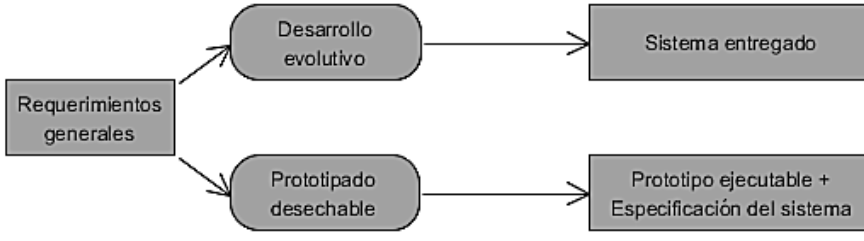



**Figura 63.** Desarrollo y prototipado incremental

Comúnmente estos softwares eran desarrollados por equipos muy grandes los cuales inclusive a veces trabajaban desde diferentes partes del mundo o también trabajaban para diferentes compañías. Un claro ejemplo de un sistema construido en períodos de tiempo largo tenemos al sistema control de aviones en el cual pueden transcurrir años desde el punto del cual se tomó las especificaciones, hasta el punto en el cual se vaya a utilizar este software. En este texto, se explica todo el trabajo que lleva desde la planificación, diseño, implementación y documentación del sistema. Pero todo este esfuerzo adicional se justifica en él

punto cuando se vaya a realizar un mantenimiento ya que se dispondrá con toda documentación necesaria para realizarla correctamente.

A pesar de todos los beneficios que tiene este tipo desarrollo en sistemas pequeños o medianos no se ven reflejados, debido a que se realizan muchos esfuerzos en la planificación dejando atrás tanto la parte de programación y pruebas. Si algún requerimiento nuevo aparece, involucraba que la documentación, especificación, diseño y el programa en sí debería cambiar.

Esto provocó que muchos de los desarrolladores se sientan insatisfechos, por lo cual propusieron nuevos enfoques en los cuales principalmente se centraban en el software antes que en el análisis y diseño. Diseñados para aquellos sistemas de negocio en los cuales los requerimientos cambian constantemente, entregando así en cada incremento un modelo o sistema totalmente funcional en el cual cliente puede interactuar y así proponer algún nuevo cambio o requisito para el sistema.

Tal vez la programación extrema sea el método ágil más conocido, en esta sección del libro se detallan las características principales que lo componen y su aplicación en el campo, pero también otros enfoques ágiles como Cristal, Serum, DSMD. La gran acogida de estos enfoques ha llevado a que se crea una integración con los enfoques más tradicionales tales como lo basados en el modelado. Dando como resultado un modelado ágil y con los principios del RUP.

A pesar de que este tipo de enfoque se centra en el desarrollo en entrega constante de incrementos, propone diversas formas de llegar a este objetivo, pero compartiendo los mismos principios. A continuación en la Tabla 3 se reflejan dichos principios.

Muchos desarrolladores que han utilizado métodos ágiles han pasado por alto sus deficiencias debido a los beneficios que contienen; pero de igual manera otros exageran los problemas de este enfoque, por tal motivo los autores DeMarco y Boehm han destacado tanto las ventajas y desventajas de los métodos ágiles, llegando a la conclusión de que un enfoque híbrido en el cual se incorpore tanto las técnicas de desarrollo y las buenas técnicas de planificación. Pero los principios de los enfoques ágiles son difíciles de realizar:

1. Una participación activa de parte de cliente, es muy beneficiosa ya que de esta dependen el éxito del proyecto, pero no siempre

| Principio | Descripción |
|---|---|
| **Participación del cliente** | Los clientes deben estar fuertemente implicados en todo el proceso de desarrollo. Su papel es proporcionar nuevos requerimientos del sistema y evaluar interacciones del sistema. |
| **Entrega incremental** | El software se desarrolla en incrementos, donde el cliente especifica los requerimientos a incluir en cada incremento. |
| **Personas, no procesos** | Se debe reconocer y explotar las habilidades del equipo de desarrollo. Se debe dejar desarrollar a los miembros del equipo sus propias formas de trabajar sin procesos formales. |
| **Aceptar el cambio** | Se debe contar con que los requerimientos del sistema cambian, por esto el sistema se diseña para dar cabida a cambios |
| **Mantener la simplicidad** | Se deben en la simplicidad tanto en el software a desarrollar como en el proceso de desarrollo. Donde esa posible se trabaja activamente para eliminar la complejidad. |

***Tabla 3.*** Los principios de los métodos agiles

es posible debido en los representantes del cliente con frecuencia no están dispuestos o tienen el tiempo para participar plenamente en el desarrollo, o no puede representar exitosamente todo los requerimientos de la empresa.

2. Es posible que en este tipo de desarrollo ágil los miembros del equipo no se logren relacionar entre sí debido a la presión que generan estos enfoques.
3. En sistemas en los cuales existen demasiados Stakeholders difícilmente se puede llegar a la conclusión, de cuáles son los requerimientos más críticos, porque cada uno tendrá diferentes prioridades.
4. Se debe mantener una simplicidad aunque esto requiere un trabajo extra, pero difícilmente se llega a estas simplificaciones debido al corto tiempo de cada incremento.

Existen otro tipo de problemas con este tipo desarrollo, tales como que el cliente utilice las organizaciones externas para la construcción del software, como ya se miran el capítulo 2. Comúnmente el documento de especificación de requerimientos es parte de contrato inicial que se da entre el cliente y el desarrollador, sin embargo, dentro de este tipo

de enfoques muy difícilmente este documento puede llegar a ser un parte del contrato debido a los constante cambios que se presentan.

Los enfoques ágiles dependen directamente de que el contrato se lo firme no estipulando los requerimientos que va a tener el software, sino más bien bajo un término de tiempo y costo para lo cual ambas partes deben estar de acuerdo. Pero esto puede provocar problemas entre cliente y el desarrollador debido a que el tiempo se puede extender por los nuevos requerimientos del cliente, genera por disputas de cuál es el culpable y quién tendrá que pagar por ello.

Como todo método tiene limitantes, por ello son los más idóneos para negocios pequeños y de mediano tamaño pero no para sistemas grandes en los cuales exista interacciones hardware y software, ni tampoco para sistemas se necesiten un análisis muy detallado de los requerimientos ya que esto puede comprometer un fallo en la seguridad o protección del sistema.

### 3.1.2 Programación extrema

Uno de los métodos más conocidos y utilizados es la Programación Extrema (XP), el cual es un método ágil, que fue nombrado por Beck debido a la participación extrema de cliente en este desarrollo Iterativo En este tipo de programación a los requerimientos se los expresa mediante historias de usuario implementadas en cada tarea. Los desarrolladores trabajan en parejas y realizan pruebas constantemente antes de concluir con el código, para que este código llegue a ser parte del sistema deben cumplir con las pruebas exitosamente. En la Figura 64 se ilustra el proceso normal XP para la producción de un incremento. En la XP se realizan varias prácticas que se resumen en la Tabla 4, las cuales se adaptan a estos principios:

1. A través de pequeños incrementos se da un desarrollo incremental, también a partir de una pequeña descripción de los requerimientos basados en la historia de usuario se puede lograr una planificación para este desarrollo
2. Los representantes del cliente debe de estar dispuesto a un compromiso a tiempo completo y también serán los responsables de que el sistema pase las pruebas de aceptación.
3. Se logra trazar un plan para el desarrollo a partir de las historias de usuarios

4. Los cambios en el sistema se dan a través de cada incremento el cual es probada continuamente para luego ser integrado en el sistema
5. En estos procesos el mantenimiento de la simplicidad se lo realiza a partir de la reestructuración del código pero sin alterar el comportamiento externo, de tal manera se obtiene mejoras del sistema constantemente.

**Figura 64.** El ciclo de entrega en la programación extrema

| Principio | Descripción |
|---|---|
| **Planificación incremental** | Los requerimientos se registran en tarjetas de historias, las historias a incluir en una entrega se determinan según el tiempo disponible su prioridad relativa. Los desarrolladores dividen estas historias en <<Tareas>> de desarrollo. Véanse las Figuras 65 y 66. |
| **Entregas pequeñas** | Las entregas del sistema son frecuentes e incrementales, añadiendo funcionalidad a la primera entrega |
| **Diseño sencillo** | Solo se lleva acabo el diseño necesario para cumplir los requerimientos actuales. |
| **Desarrollo previamente probado** | Se utiliza un sistema de pruebas de unidad automatizado para escribir pruebas para nuevas funcionalidades antes de que se implementen. |
| **Refactorización** | Se espera que todos los desarrolladores refactoricen el código continuamente tan pronto como encuentren posibles mejoras en el código. Esto conserva el código sencillo y mantenible. |
| **Programación en parejas** | Los desarrolladores trabajan en parejas, verificando cada uno el trabajo del otro y proporcionando la ayuda necesaria para hacer siempre un buen trabajo. |
| **Propiedad colectiva** | Las parejas de desarrolladores trabajan en todas las áreas del sistema de modo que no se desarrollen islas de conocimiento y todos los desarrolladores posean todo el código. Cualquiera puede cambiar cualquier cosa. |
| **Integración continua** | En cuanto acaba el trabajo de una tarea se integra en el sistema entero. Después de la integración se debe aplicar al sistema todas las pruebas de unidad. |
| **Ritmo sostenible** | No se considera aceptables grandes cantidades de horas extras, ya que a menudo el efecto que tienen es que se produce la calidad del código y la productividad a medio plazo. |
| **Cliente presente** | Debe estar disponible al equipo un representante de los usuarios finales del sistema a tiempo completo. El cliente es miembro del equipo de desarrollo y responsable de formular al equipo de requerimientos del sistema para implementación. |

En la programación Extrema el cliente se encuentra constantemente implicado en el establecimiento de los requerimientos críticos, llegan a ser parte del equipo del desarrollan y discuten con los demás integrantes. Se unen y crean juntos las historias de usuario en las cuales se especifican sus necesidades. Todo el equipo tratara de construir un incremento a partir de esta historia de usuario. En la Figura 3.4 se muestra un claro ejemplo de la forma de una historia de usuario para un sistema.

| **Descarga e impresión de un Articulo** |
| --- |
| En primer lugar, seleccione el artículo que desea de una lista visualizada. Tiene que decir al sistema como va a pagar - se puede hacer a través de una suscripción, una cuenta de empresa o mediante una tarjeta de crédito.<br><br>Después de esto, obtiene un formulario de derechos de autor del sistema para que lo rellene. Cuando haya enviado se descarga el artículo en su computadora.<br><br>Elija una impresora y se imprimirá una copia del artículo. Luego el sistema responde que la impresión se ha realizado correctamente.<br><br>Si es un artículo de solo impresión, no puede guardarla versión de PDF, por lo que automáticamente se elimina de su computadora. |

**Figura 65.** Tarjeta de historia para la descarga de documentos.

Una vez determinadas las tarjetas de historias de usuario, el equipo las divide en esfuerzo, recursos y tiempo requerido para realización de este; después de esto el representante del cliente identifica cuáles son las historias de mayor importancia para el negocio y posteriormente pasan a su implementación. Obviamente cuando los requerimientos cambian, las historias que no hayan sido implementadas a un pueden ser eliminados o en su caso modificadas. En casos de que el sistema ya fue entregado y necesita cambios se realizaran nuevas historias las cuales pasaran por el mismo proceso explicado anteriormente.

En este tipo de procesos se llegan a realizar muchas versiones software y todos los incrementos resultantes comúnmente se los entrega en el término de 2 meses, luego de haber aprobado exitosamente cada una de las pruebas automatizadas.

En los conceptos tradicionales se estipula que se le diseñan para prever los cambios para que así se pueda implementar fácilmente. Pero en la programación extrema se descarta estos puntos porque a pesar

de prepararse para estos cambios surgirán nuevos cambios, por lo cual se lo considera una pérdida de tiempo y esfuerzo.

En la programación extrema se refactoriza el código en cada incremento para que no exista un desgaste en la estructura del software, es decir, que con esta técnica constantemente se hacen cambios en la parte interna de la aplicación pero sin dañar son estructura externa.

#### 3.1.2.1 Pruebas en XP.

Como ya se mencionó al principio de este capítulo las diferencias que define tanto al desarrollo iterativo del que está basado en las especificaciones, es la forma en cómo se realizan las pruebas del sistema, al no existir un documento de especificación es muy difícil que un equipo externo realice pruebas en el sistema, por tal motivo, la fase de pruebas en estos enfoques suelen ser informales. La programación extrema se centra principalmente en los proceso de pruebas, construidos de tal manera que reduzca la aparición de nuevos errores en los siguientes incrementos. A continuación se indican las características claves de la Programación Extrema:

1. Que la implementación haya sido probada antes de integrarla al sistema.
2. Creación de las pruebas a partir del escenario.
3. Implicar al usuario en las fases de pruebas y validación del sistema.
4. Realizar pruebas automatizadas que agilicen el desarrollo.

Al desarrollar desde un principio las pruebas que se realizaran en el sistema, implícitamente se definen también una interfaz y especificación del funcionamiento. En la Programación Extrema, las historias de usuarios son divididas en tareas que representan algunas de las características del sistema. En la Figura 66 se muestra algunas de las task cards, las cuales fueron creadas a partir historia de usuario (Figura 65). Cada una de las tareas genera pruebas de unidad que sirven para verificar la fase de implementación. Por ejemplo en la Figura 67, se ve la descripción de un caso de pruebas.

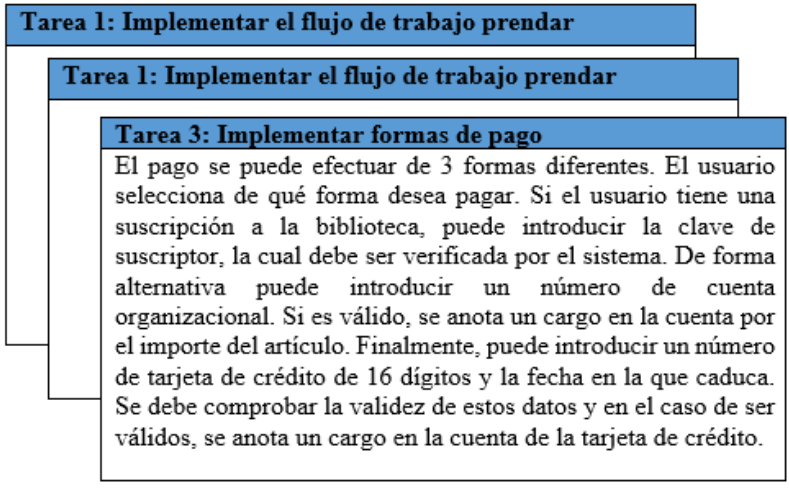



**Figura 66.** Tarjetas de tareas para la descarga de documentos**.**

**Tarea 4: Pruebas de validez de la tarjeta de crédito**

**Entrada:**

Una cadena que representa el número de tarjetas de créditos y dos enteros que representa el mes y el año de la caducidad de la tarjeta.

**Pruebas:**

Comprobar que todos los bytes de la cadena de dígitos.
Comprobar que el mes se encuentra entre 1 y 2 y que el año es mayor o igual que el año actual.
Utilizando los 4 primeros dígitos del número de la tarjeta de crédito, comprobar que el emisor de la tarjeta es válido consultando la tabla de emisores de tarjetas.
Comprobar la validez de la tarjeta de crédito enviando el número de tarjeta y la fecha en la que se caduca el emisor de la tarjeta.

**Salidas:**

Mostrar un mensaje indicando que la transacción ha sido correcta o indicando que la tarjeta no es válida.

***Figura 67.*** Validez de la tarjeta de crédito.

Al desarrollar pruebas con anterioridad y usar las pruebas ya estandarizadas se demuestra las principales virtudes de la Programación Extrema. Estas pruebas deberían ser creadas independientemente de la aplicación, para que simulen el tipo de entradas en el sistemas verificando si cumple con lo que se indicó en las especificaciones. Dentro

de este desarrollo cada vez que se agregue una nueva funcionalidad se realizan pruebas para detectar problemas y visualizar si se están cumpliendo los requerimientos previamente establecidos.

Las personas encargadas de crear las tareas deben ser capaces crearlas de tal manera que sean claras para los desarrolladores evitando la ambigüedad; así se evitan los retrasos de las pruebas, ya que el ritmo con el que trabajan los desarrolladores es muy elevado en comparación del equipo encargado de las pruebas, por lo tanto, deben tener claro el funcionamiento que debe tener e sistema. En ciertos casos no se pueden realizar pruebas de unidad para una sección ya que resulta muy difícil implementarlo o también no se logra abarcar con estas pruebas todas las posibles situaciones excepcionales que puedan suceder.

#### 3.1.2.2 Programación en parejas

Una característica que las destaca de otras metodologías es la programación en parejas; en la que ambos trabajan en un misma sección del programa, pero esto no quiere decir que siempre tendrán la misma parejas, sino más bien busca que el equipo se interrelacione con los demás miembros del equipo. Las principales ventajas que nos brindan son:

1. Una de las principales ventajas es que el equipo es dueño del software, por lo tanto no se busca un culpable al tener problemas o errores con el código, en cambio, se fomenta la responsabilidad y el trabajo en equipo para resolver estos problemas.
2. La programación en pareja también se puede tomar como una revisión informal del código, en el cual se pueden detectar un gran porcentaje de los errores del software en construcción y resultar ser un proceso mucho más económico que realizar inspecciones formales.
3. Además no ayuda a una restructuración del código fuente, cambiando su estructura interna en cada incremento mejorando su claridad. El problema de implementar refactorización en otro entorno desarrollo es un esfuerzo innecesario que solo se obtendría resultado a largo plazo, mientras que en XP se obtienen beneficio mutuo entre todos los miembros del equipo

A simple vista se podría decir que existe menos productividad y eficiencia en la programación en pareja, pero se ha comprobado que esto no es cierto, ya que al trabajar en pareja, se discute sobre el software antes de implementarlo de tal manera se evita tener comienzos en falso, rehacer nuevamente el código y así pasar menos tiempo corrigiendo los errores.

### 3.1.3 Desarrollo rápido de aplicaciones

El desarrollo rápido de aplicaciones (RAD), son unas herramientas que nos permite crear, buscar, ver y presentar datos en informes. En la Figura 68se puede observar a un sistema RAD. Las herramientas que se incluyen en un entorno RAD son:

1. Los Lenguajes de programación de base de datos, que nos permiten la inclusión de comando SQL directamente a partir de formularios rellenados por el usuario final.
2. Generadores de interfaces, los cuales nos permiten incluir sus componentes y visualización de sus datos.
3. Enlaces a aplicaciones de oficina, tales como los procesadores de textos y hojas de cálculos, los cuales nos permiten analizar y manipular información para generar modelos de informes.

El desarrollo rápido de aplicaciones, ha llegado a tener mucho éxito debido a lo anteriormente visto en el Capítulo 1. Los sistemas RAD, están enfocados a la construcción de software interactivo, por medio del cual los usuarios finales en su terminal obtienen todos los cambios realizados de la base de datos organizacional.

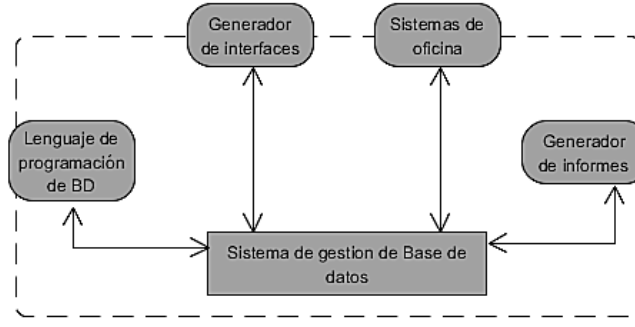



**Figura 68.** Un entorno de desarrollo rápido de aplicaciones

Muchos de los sistemas de negocio, se asisten a partir de formularios muy elaborados para entrada y salida de datos; llegando a ser un gran recurso debido a nos ayudan a generar pantallas e informes. A los formularios vinculados se los define como pantallas, los cuales deben proporcionar:

1. 1Definición de formularios interactivos, nos ayuda definir a los componentes de la manera que los queramos visualizar u organizar dentro del sistema.
2. Vinculación de los formularios, permite que el desarrollador detalle cuáles serán las entrada que nos ayudaran a visualizar otras pantallas
3. Verificación de campos, ayuda a definir al desarrollador los parámetros de entradas, válidos para los campos de los formularios.

Hoy en día existen muchos entornos RAD enfocados en base de datos en navegadores, los cuales le permite al usuario acceder a internet desde cualquier lugar. Ayudando a la reducción de los costos pero tiene como limitante el soporte de los protocolos y navegadores de internet.

Entornos de desarrollo visual tales como Visual Basic, nos permite crear este tipo de aplicaciones muy fácilmente a través de la reutilización, ya que nos permiten diseñar software a través del arrastre de botones, campos, pantallas, entre otros. Otra ventaja es que estamos constantemente visualizando como va quedando el interfaz del sistema.

Este enfoque se ve ilustrado en la Figura 69, donde se pude observar una pantalla con sus componentes, como lo son campos y botones. Gracias a este tipo de programación visual se puede establecer componentes reutilizables. También se puede observar en la Figura 69 la asociación de estos componentes con algunos otros elementos.

Otros lenguajes como Visual Basic son ejemplos sofisticado de lenguajes de creación de secuencias los cuales incluyen estructuras de control que reducen el tiempo que se necesita para el desarrollo.

Los sistemas bajo estos enfoques permiten desarrollar rápidamente y de forma sencilla, ya que utiliza herramientas que facilitan la construcción del mismo. Muy a menudo, en los lenguajes de creación de secuencias existen ciertas dependencias entre algunas partes del sistema, lo que provoca problemas al requerir algún cambio.

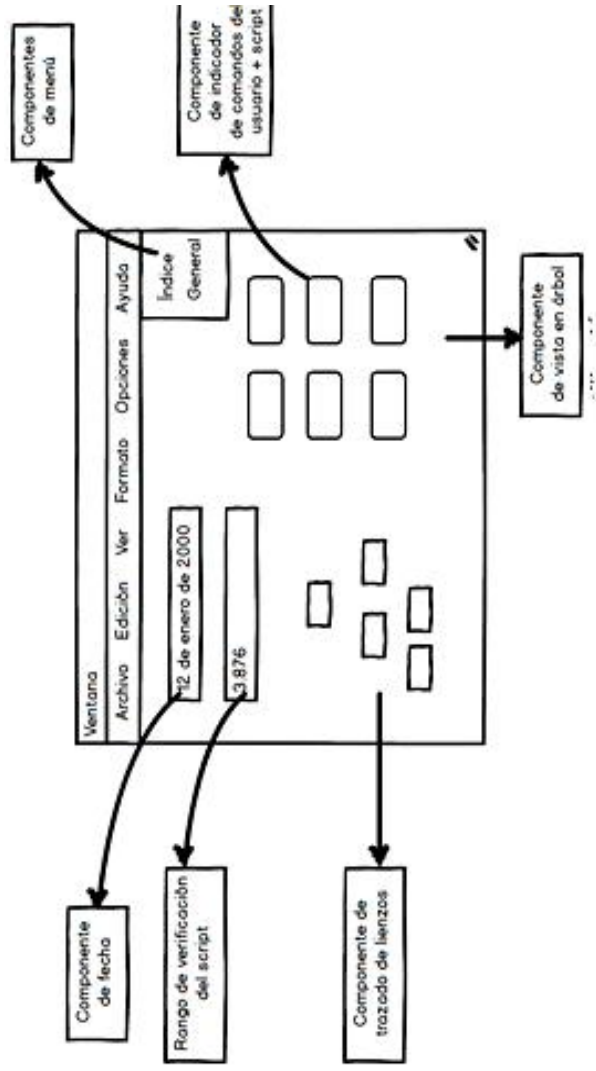



**Figura 69.** Programación visual con reutilización

Los RAD utilizan la integración de componentes reutilizables, basándose en COTS (Commercial Off-the-Shelf), que no es más que utilizar software que ya existe en el mercado para funcionalidades que necesite el sistema que estemos desarrollando; por ejemplo si el sistema necesitase un procesador de texto podríamos usar cualquiera ya existente para acoplarlo a nuestro sistema. En este capítulo, se estudia con mayor detalle los COST.

Los entornos desarrollados en COTS, nos permite construir una aplicación con todas sus funcionalidades. Para entender estos enfoques podemos ver la manera en que los datos son procesados y como son organizados estos documentos en un contenedor, los cuales pueden estar compuestos por objetos, pero conteniendo distintos datos que comúnmente se enlazan y organizan para que la aplicación sea iniciada al ingresar a un objeto.

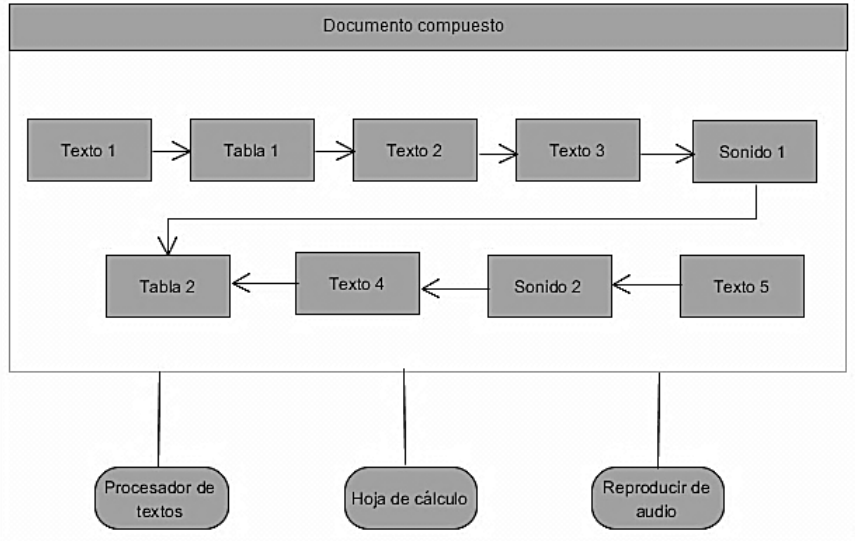



***Figura 70.*** Vinculación de aplicaciones

En la Figura 70 se muestra un sistema formado por archivos de texto, cálculo y de audio. Un claro ejemplo de esto es cuando abrimos un archivo de audio inmediatamente se abre el reproductor de audio. En palabras técnica al dirigirnos a un objeto en particular el sistema

accederá a la función más idónea o asociada para la lectura de este tipo de archivos. Las ventajas de estos enfoques es que son de muy bajos coste, y también al usar aplicaciones ya construidas el usuario puede estar familiarizado con este tipo de aplicaciones.

### 3.1.4 Prototipado del software

Los prototipos, no son más que versiones de la aplicación en construcción que comúnmente nos sirven para comprobar el su funcionamiento, identificar problemas y crear soluciones. Es tipo de desarrollo incremental es muy útil, porque se puede interactuar desde las primeras etapas del desarrollo. Estos prototipos se los puede utilizar de muchas maneras dentro de un proceso:

1. Los prototipos nos permiten validar los requerimientos del sistema.
2. Dentro del proceso del diseño, nos permiten encontrar nuevas soluciones a un problema en concreto.
3. Dentro de la fase de pruebas, son de mucha utilidad ya que se utiliza al prototipo para realizar pruebas como si fuere el sistema final a entregar.

A los prototipos también se los puede utilizar para comprobar si el diseño construido es funcional y eficiente, ya que no es suficiente tener descripciones contextuales del funcionamiento o diagramas del sistema para construcción del software.

El problema primordial del proceso de las pruebas es validarlas para obtener los resultados que se esperan de ellas. En la Figura 71 se muestra las pruebas Back – to – Back, en donde se envían exactamente los mismo casos de pruebas al prototipo y al sistema. La forma en que se detectan un problema es a partir de los resultados. Si presentan diferencias en sus resultados existe algún problema, caso contrario los sistemas van de acuerdo a como se han especificado en el funcionamiento de las pruebas. En un conjunto de estudios realizados se observó los siguientes beneficios:

- Aumenta la usabilidad del sistema
- Mejora la interacción entre el sistema y el usuario final
- Aumenta la calidad del diseño de la interfaz
- Ayuda a mejoras en el mantenimiento
- Se reduce el esfuerzo en el desarrollo

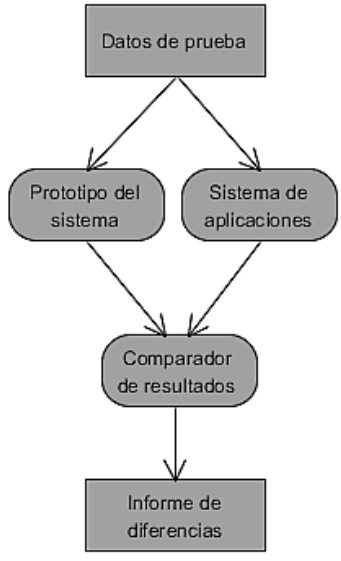



***Figura 71.*** Pruebas Back-to-back.

En la Figura 72 muestra un modelo del proceso para el desarrollo de prototipos, cuyos objetivos son: desarrollar sistemas de validación y viabilidad del sistema. Luego de esta etapa pasa a un proceso en el cual se decidí que incluir en el sistema evaluando los tiempos de respuesta.

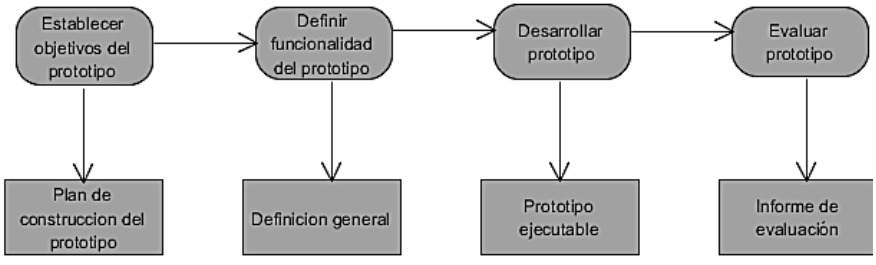



***Figura 72.*** Proceso de desarrollo de prototipos.

En la etapa final se evalúa al prototipo, se debe predecir la interacción del usuario con el sistema y prever que el usuario debe acostumbrase con el nuevo sistema.

El principal problema de los prototipos desechables es que muchas veces el modo en el que se utiliza estos no es igual al sistema final. El encargado de las pruebas no sabe ser el usuario típico del sistema, en estos casos existen muchas presiones por parte de los administradores para entregar un prototipo desechable, lo que provoca la entrega de sistemas de muy baja calidad, pero esto no es aconsejable por las siguientes razones:

1. Es muy difícil hacer cumplir los requisitos no funcionales en los prototipos desechables dejando a un lado los aspectos de rendimiento, protección, robustez y fiabilidad.
2. Debido a que se entrega un prototipo desechable, no existe documentación en las especificaciones del diseño, provocando insuficiente información para lograr un mantenimiento en un periodo posterior.
3. En estos prototipos tampoco se respeta la calidad del producto ya que se busca sacar un pequeño incremento en un periodo corto.

Los prototipos desechables no tienen que ser completos, es decir, respetar estándares de calidad. En el Capítulo 2, se establece como ayudar al usuario construir interfaces a través de las historias de usuario.

## 3.2. Reutilización del software.

### 3.2.1 El campo de la reutilización.

A lo largo de los últimos veinte años, se han expuesto numerosas técnicas para sostener la reutilización del software. Esta reutilización se puede hacer a distintos niveles (desde funciones simples a aplicaciones complejas), además las normas para elementos reutilizables favorecen la reutilización.

Una vez analizada esta lista de técnicas para reutilización, el para saber ¿Cuál de estas técnicas es la más conveniente? Evidentemente, esta elección va obedecer a los requisitos del sistema a desarrollar, la tecnología, elementos reutilizables libres y la destreza del grupo de desarrollo. Los componentes esenciales que se deberían tener en cuenta

al momento de programar la reutilización son:

1. La agenda de desarrollo del software. Si el software tiene que desarrollarse de forma rápida, debería procurarse reutilizar sistemas comerciales en vez de componentes individuales. Éstos son elementos reutilizables de grano grueso. Pese al cumplimiento de todos estos requisitos, el proceso puede no ser perfecto, pero esta aproximación disminuye la cantidad de desarrollo requerido.
2. Vida esperada del software. Si se está trabajando en un sistema de larga duración, habría que centrarse en el mantenimiento del sistema. En esas condiciones, no solamente correspondería pensarse en las posibilidades inmediatas de la reutilización, sino que también en las oposiciones a extenso plazo. Se tendrá que acomodar el sistema a ajenos requerimientos, lo cual posiblemente simbolice crear cambios a los componentes y cómo éstos son utilizados. Si no se tiene acceso al código fuente, probablemente correspondería evitarse utilizar componentes y sistemas de proveedores del exterior; no se logra estar seguro de que estos proveedores serán competentes de continuar aguantando el software reutilizado.
3. Las habilidades, conocimientos, y uso del grupo de proceso. Todas las tecnologías de reutilización son muy complejas y se precisa suficiente tiempo para entenderlas y utilizar de forma efectiva. Sin embargo, si el grupo de proceso posee destrezas en un área exclusiva, posiblemente habría que concentrarse en ella.
4. La criticidad del software y sus imposiciones no funcionales. Para un sistema crítico que tiene que ser garantizado por un regulador exterior, se tiene que hacer un caso de confianza para el sistema. Esto es dificultoso si no se tiene paso al código origen del software. Si el software tiene imposiciones de rendimiento estrictos, puede ser inadmisible utilizar habilidades tales como reutilización a través de creadores de programas. Estos sistemas tienden a crear código relativamente inútil.
5. El mando de las aplicaciones. En algunos mandos de aplicaciones como los sistemas de información médica y de fabricación,

hay varios bienes genéricos que pueden reutilizarse para su configuración a una situación exclusiva. Si se está trabajando en aquellos dominios, persistentemente deberían reflexionarse sobre éstos como una elección.

6. La plataforma sobre la que el sistema se va a elaborar. Ciertos modelos de elementos, como COM/Active X, son plataformas determinadas de Microsoft. Si se está desplegando sobre una plataforma como éstas, este acercamiento puede ser la más ajustada. De igual manera, los sistemas de aplicaciones genéricos logran ser plataformas determinadas y simplemente es posible reutilizarlos si el sistema se traza para la misma plataforma. El total de sistemas de reutilización útiles es tal que, en la totalidad de las situaciones, consta la posibilidad de cualquiera reutilización del software. Si se consigue o no la reutilización es muy seguido una cuestión de gestión más que una cuestión técnica. Los gestores consiguen ser reacios a comprometer sus requerimientos para consentir el uso de elementos reutilizables, o consiguen decidir que el proceso de componentes originales lograría ayudar a establecer una base de activos software. Es viable que no comprendan los peligros asociados con la reutilización tan bien como entienden los peligros del proceso original, sin embargo, si los peligros de un nuevo proceso software logran ser más elevados, ciertos gestores pueden preferir los peligros conocidos a los desconocidos.

### 3.2.2 Patrones de diseño.

El diseñador que pretende reutilizar elementos ejecutables, se limita manera inevitable por las medidas de diseño que han sido impuestas por los implementadores de esos elementos. Los cuales han incluid muchas de la veces desde algoritmos exclusivos que han sido manejados para implementar los elementos hasta los tipos y objetos en las interfaces de los elementos.

Cuando las medidas de diseño están en pugna con sus requerimientos distintivos, reutilizar el elemento es improbable o introduce ineficiencias en su sistema. Una manera de solucionar esto, es reutilizar bocetos que no incluyen referencias de la implementación, esto le permite al diseñador alcanzar a implementarlos para concentrarse en

las exigencias exclusivas de la aplicación. Las instancias iniciales de para la reutilización se originaron en la documentación y publicación de algoritmos esenciales y más tarde en archivos de tipos abstractos de datos tales como pilas, listas y árboles. Últimamente, la reutilización ha sido incluida en los modelos de diseño.

Los modelos de diseño se provinieron de las ideas pronunciadas por Christopher Alexander, quien insinuó que existían algunos modelos del diseño de edificios que eran frecuentes e inherentemente atractivos y efectivos. Se considera como patrón a una representación del problema y la idea de su solución, de forma que se logre reutilizar en diversos contextos. El patrón no es una descripción detallada de una solución ajustada a una dificultad común.

Gran parte de los diseñadores piensan en los modelos de diseño como una manera de sobrellevar el diseño orientado a objetos, tomando en cuenta detalles de los tales como la herencia y el polimorfismo para proporcionar generalidad. El principio general del diseño es la encapsulación considerada como experiencia en un patrón es que es equivalentemente aplicable a todos los acercamientos de diseño software.

Gamma y terceros definen los cuatro elementos fundamentales de los patrones de diseño:

1. 1Un nombre que es una reseña significativa del patrón.
2. Una representación del área de dificultad que explica cuándo logra aplicarse el patrón.
3. Una representación de las partes de la solución del esbozo, sus relaciones y sus responsabilidades. Ésta no es una representación específica de diseño. Es una plantilla para una solución del bosquejo que puede instanciarse de diferentes maneras. A menudo ésta se expresa gráficamente y tipos las relaciones entre los objetos y las clases de las cosas en la solución.
4. Una afirmación de las consecuencias las deducciones y compromisos de aplicar el patrón. Esto puede auxiliar a los diseñadores a percibir si un patrón puede ser aplicado de forma segura en una situación particular.

Actualmente están existen un dominante número de patrones publicados que abarcan diferentes dominios de aplicaciones y lenguajes. El conocimiento de un patrón como un concepto reutilizable ha sido

desplegado en varias áreas además del esbozo del software, que incluye gestión de clasificaciones, diseño de interfaces de usuario y escenarios de interacciones.

### 3.2.3 Reutilización asentada en productores.

El significado de reutilización mediante modelos tiene que ver con la representación del concepto de una manera abstracta y dejando al desarrollador la labor de fundar una implementación. Un acercamiento alternativa a ésta es la reutilización asentada en productores. En este acercamiento, el conocimiento reutilizable se captura en un sistema productor de programas que puede ser programado por especialistas en el dominio manejando un lenguaje orientado a dominios o una herramienta CASE interactiva que soporte la generación de sistemas.

La reutilización asentada en productores se aprovecha del hecho de que las aplicaciones de igual dominio, tales como sistemas de negocio, tienen arquitecturas frecuentes y realizan funciones parecidas.

La reutilización asentada en productores ha tenido conquista sobre todo para sistemas de aplicaciones de negocios, y existen varios productos generadores de aplicaciones de negocios utilizables. Estos pueden crear aplicaciones completas o pueden computarizar parcialmente la creación de aplicaciones y dejar que el programador las cumpla con datos definidos. El acercamiento a la reutilización asentada en productores también se utiliza en otras áreas, incluyendo:

1. Generadores de examinadores para el proceso del lenguaje. La entrada del generador es una gramática que detalla el lenguaje que va a ser examinado, y la salida es el examinador del lenguaje.
2. Generadores de código en herramientas CASE. El ingreso de estos generadores es un bosquejo de software y la salida es un programa que implementa el sistema trazado. Pueden basarse en modelos UML y, dependiendo de la información en los modelos UML, generar un programa completo o elemento, o bien un esqueleto de código.

La reutilización asentada en generadores es rentable para aplicaciones tales como proceso de datos de negocios. Es mucho más sencillo para los usuarios finales desarrollar programas utilizando generadores frente a otras proximidades basadas en elementos para la reutilización. Sin

embargo, inevitablemente hay deficiencias en los programas generados. Esto significa que puede ser imposible utilizar esta cercanía en sistemas con requerimientos de elevado rendimiento.
La programación generativa es un elemento clave de técnicas emergentes de progreso de software que combina la generación de programas con el progreso basado en elementos.

Una de las proximidades con más renovaciones, es el progreso de software encaminado a características (AOSD). El progreso de software, encaminado a características aborda uno de las mayores dificultades en el bosquejo del software es el problema de la separación de intereses. La separación de intereses es un principio de bosquejo básico; debería trazarse el software.

En la programación orientada a características, los intereses compartidos se implementan como características dentro del programa, se define dónde se debería asociar un aspecto; esto se denominan puntos de enlace. Las características se desarrollan de forma apartada; a continuación, en un paso de pre compilación denominado entrelazado de características, son enlazados mediante los puntos de enlace. El entrelazado de características es una forma de generación de programas; la salida del proceso de entrelazado es un programa en el que se ha integrado el código del aspecto.

### 3.2.4 Marcos de trabajo de aplicaciones.

Las primeras personas que propusieron el progreso encaminado a objetos sugirieron que los objetos eran la abstracción más adecuada para la reutilización, sin embargo, la práctica ha demostrado que los objetos son a menudo de granos muy finos y excesivos especializados para una aplicación exclusiva.

Un marco de trabajo es un bosquejo de un subsistema formado por una colección de clases concretas, abstractas y la interfaz entre ellas. Los detalles exclusivos del subsistema de aplicación son ejecutados añadiendo elementos y proporcionando implementaciones concretas de las clases abstractas en el marco de trabajo. Los marcos de trabajo raramente son aplicaciones por sí mismas. Las aplicaciones se edifican regularmente integrando varios marcos de trabajo. Fayad y Schmidt detallan tres clases de marcos de trabajo:

1. Marcos de trabajo de infraestructura de sistemas. Estos mar-

cos de trabajo soportan 4 procesos de infraestructuras de sistemas tales como comunicaciones, interfaces de usuario y compiladores.

2. Marcos de trabajo para la integración de middleware. Consisten en un conjunto de estándares y clases de objetos asociados que soportan la comunicación de elementos y el intercambio de información.
3. Marcos de trabajo de aplicaciones empresariales. Se refieren a dominios de aplicaciones definidos tales como telecomunicaciones o sistemas financieros. Estos marcos de trabajo encapsulan el conocimiento del dominio del api caución y soportan el progreso de aplicaciones para los usuarios finales.

Tal y como el nombre indica, un marco de trabajo es una estructura genérica que puede ser extendida para hacer un subsistema o aplicación más específico. Este es implementado como una colección de clases de objetos concretas y abstractas. Para ampliar el marco de un trabajo, se tienen que añadir clases concretas que hereden operaciones de las clases abstract en el marco, además se deben definir callbacks. Los callbacks son métodos que se llama como réplica a programas reconocidos por el marco de trabajo.

Las aplicaciones construidas manejando marcos de labor pueden ser las bases para una posterior reutilización a través del concepto de líneas de productos software o familias de aplicaciones, tal y como se indica en la Sección 3.2.5.2. Debido a que estas aplicaciones se construyen utilizando un marco; se simplifica la modificación de miembros de la familia para hacer desconocidos miembros.

El problema fundamental con los marcos de trabajo es su complejidad inherente y el tiempo que lleva aprender a utilizarlos. Pueden requerirse varios meses para comprender totalmente el marco de trabajo, por lo que es muy probable que, en organizaciones grandes, algunos ingenieros software se conviertan en especialistas en marcos de trabajo. No hay duda de que ésta es una cercanía efectiva para la reutilización, pero es muy elevado el coste que presume introducirla en los procesos de progreso del software.

### 3.2.5 Reutilización de sistemas de aplicaciones.

La reutilización de sistemas de aplicaciones envuelve reutilizar siste-

mas de aplicaciones completos configurando un sistema para un entorno determinado o integrando dos o más sistemas para crear una nueva aplicación. Tal y como se indicó en la Sección 3.2.1. La reutilización de sistemas de aplicaciones es a menudo la habilidad de reutilización más segura, implica la reutilización de activos de grano grueso que pueden ser rápidamente configurados para hacer un desconocido sistema.

En esta sección, se analizan dos tipos de reutilización de aplicaciones: la creación de desconocidos sistemas integrando dos o más aplicaciones comerciales y el progreso de líneas de servicios. Una línea de productos es un conjunto de sistemas basados en una arquitectura base común y elementos compartidos. El sistema base se traza de forma determinada para ser configurado y adaptado a fin de adecuarse a las necesidades determinadas de los desiguales clientes del sistema.

#### 3.2.3.1 Reutilización de productos COTS.

La denominación producto COTS se aplica a un sistema software que puede manejarse sin cambios por su comprador. Implícitamente todo el software de sobremesa y un gran número de productos servidores son software COTS. Este software se traza para uso general, regularmente incluye varias características y funciones para que sea potencialmente reutilizable en heterogéneas aplicaciones y entornos.

Muy pocos desarrolladores podrían tomar en consideración la implementación de su propio sistema de gestión de base de datos; sin embargo, hasta mediados de los 90 simplemente existían unos cuantos sistemas grandes, como los sistemas de gestión de bases de datos y monitores de teleproceso, que fueran manejados de forma rutinaria. La mayoría de los sistemas grandes se trazaba como sistemas únicos y a menudo había muchas dificultades en continuar que estos sistemas, trabajasen de forma conjunta.

En la actualidad, es natural para los sistemas grandiosos el poseer definidas interfaces <Programación de Aplicaciones (APIs) que acceden programar el acceso a las funciones  COTS ofrecen, es posible reducir costes y tiempos de entrega en varios órdenes de magnita comparados con el progreso de desconocido software. Además, los riesgos pueden reducirse ya que el producto se encuentra disponible y los gestores pueden ver si dicho producto satisface si requerimientos.

Para desarrollar sistemas utilizando productos COTS, se tienen que

tomar varias elecciones de bosquejo:

1. ¿Qué productos COTS ofrecen la funcionalidad más adecuada? Si todavía no se tiene practica con un producto COTS, puede ser dificultoso decidir qué producto es el m adecuado.
2. Para el intercambio de datos los productos autónomos utilizan estructura de datos los cuales para ser interpretados necesitan de adaptadores.
3. ¿Qué características de un producto se utilizarán realmente? La mayoría de los por duelos COTS poseen más funcionalidades de las que se requieren y las funcionalidades a menudo se duplican entre desiguales productos. Se tiene que decidir qué características de qué productos son las más ajustadas para los propios requerimientos. Si es posible, debería también denegarse el acceso a funcionalidades no utilizadas debido a que pueden interferir en el funcionamiento natural del sistema. El fallo del primer vuelo del cohete Ariane 5, comentado en el Capítulo 4, se debió a un fallo en una funcionalidad no utilizada de un subsistema manejado.

La empresa ya posee un sistema de pedidos que es manejado por el departamento de adquisiciones. Éste ya está integrado con sus sistemas de facturación y reparto. Para hacer el desconocido sistema de pedidos, se integra el sistema antiguo con una plataforma de comercio electrónico basada en web y un sistema de correo electrónico que gestiona las comunicaciones con los usuarios.

El comercio electrónico tiene su propio formato para los pedidos, conformidades con las entregas, y así sucesivamente, los cuales tienen que acoplarse al formato manejado por el sistema de pedidos. En el sistema de comercio electrónico está integrado el sistema de correo electrónico para enviar las notificaciones a los usuarios, pero el sistema de pedidos nunca fue trazado para esto, por lo tanto, tiene que desarrollarse otro adaptador para convertir las notificaciones en mensajes de correo electrónico.

En principio, el uso de un sistema COTS a gran escala es el mismo que el uso de cualquier otro elemento más específico. Se tienen que comprender las interfaces del sistema y utilizarlas exclusivamente para comunicarse con el elemento. Se tiene que buscar un equilibrio entre los requerimientos definidos, el progreso rápido y la reutilización. Se debe que trazar una arquitectura del sistema que permita que los

sistemas COTS operen conjuntamente. Sin embargo, existen dificultades con la integración de sistema.

1. Ausencia le control sobre la funcionalidad y el rendimiento. Si bien la interfaz que presenta un producto puede parecer que oferte las facilidades requeridas, éstas pueden no estar implementadas adecuadamente o pueden tener un pobre rendimiento. El producto puede tener operaciones ocultas que interfieran en su funcionamiento en una situación determinada. Solucionar estos problemas debe ser una ventaja para el integrador de la utilidad COTS, sino no es un problema que corresponda al vendedor del producto. Los usuarios simplemente tienen que buscar la forma de sortear los problemas si quieren reutilizar el producto COTS.
2. Conflictos con la interoperabilidad del sistema COTS. Cualesquiera veces es difícil lograr efectos COTS que trabajen de forma unida debido a que cada producto se basa en sus propias suposiciones de cómo debería manejarse. Garlan y otros publicaron su práctica en el intento de integración de cuatro servicios COTS, y observaron que tres de estos productos estaban basados en programas, pero que cada uno usaba un modelo diferente de programas y consideraron que cada uno de dichos modelos tenía acceso exclusivo a la cola de programas.
3. No existe control sobre la evolución del sistema. Los vendedores de productos COT toman sus propias decisiones sobre los cambios en el sistema como réplica a las funciones del mercado. Concretamente, en productos para PCs se producen frecuente mente nuevas versiones y pueden no ser compatibles con todas las versiones antepuestas. Las nuevas versiones pueden tener funcionalidades adicionales no deseadas, versiones previas pueden dejar de estar disponibles y no tener soporte.
4. Soporte de los vendedores de servicios COTS. El nivel de soporte disponible los proveedores de COTS es significativo cuando surgen dificultades debido a que (desarrolladores no poseen acceso al código fuente y a la registro detallada del sistema.

Desde luego, no es probable que todas estas dificultades ocurran en cada caso, pero al menos una de ellas se va a producir en la mayoría de los proyectos de composición de COTS. En consecuencia, los beneficios

en coste y tiempo derivados de la reutilización de COTS son menores que los que podrían esperarse en un principio.

En muchos casos, el coste de mantenimiento y evolución de un sistema puede ser mayor cuando se utilizan productos COTS. Todas las dificultades antepuestas son dificultades del ciclo de vida; no sólo afectan al progreso inicial. En un sistema, cuantos menos desarrolladores originales haya, más personas estar implicadas en su mantenimiento, por lo que es más probable que se planteen dificultades con la integración de productos COTS.

A pesar de estas dificultades, los beneficios de la reutilización de productos COTS son significativos debido a que estos sistemas ofrecen mucha más funcionalidad a la persona que los reutiliza. Pueden ahorrarse meses y a veces años de esfuerzo de implementación y los tiempos de progreso del sistema se reducen drásticamente

**3.2.5.2 Líneas de productos software.**

Una de las proximidades más efectivas para la reutilización es la creación de líneas de productos software o familias de aplicaciones. Una línea de productos es un conjunto de aplicaciones con una arquitectura común determinada de dichas aplicaciones, tal y como se expuso el Capítulo 3. Cada aplicación determinada se especializa de alguna manera. El núcleo como de la familia de aplicaciones se reutiliza cada vez que se requiere una nueva aplicación. El nuevo progreso puede implicar una configuración determinada de elementos, implementación de elementos adicionales y adaptación de algunos elementos para satisfacer las demandas.

Existen diferentes formas de realizar una especialización para hilera de productos:

1. Especialización de la plataforma. Se desarrollan versiones de la aplicación para desiguales plataformas.
2. Especialización del entorno. Se crean versiones de la aplicación para gestionar entornos operativos y dispositivos periféricos concretos.
3. Especialización de la funcionalidad. Se crean versiones de la aplicación para clientes definidos que tienen desiguales requerimientos.
4. Especialización del proceso. El sistema se adapta para tratar

con procesos de negocio definidos.

Las líneas de productos software se trazan para ser reconfiguradas. Esta reconfiguración puede implicar añadir o eliminar elementos del sistema, definir parámetros y restricciones para los elementos del sistema, e incluir conocimiento de los procesos de negocio. Las líneas de productos software pueden ser configuradas en dos puntos del proceso de progreso:

- Configuración durante el despliegue, en donde un sistema genérico se traza para su configuración por un cliente o consultores que trabajan con el cliente. El conocimiento de los requerimientos definidos del cliente y el entorno del sistema operativo se incluye en un conjunto de ficheros de configuración que son manejados por el sistema genérico.
- Configuración durante el bosquejo, en donde la organización que está desarrollando el software modifica un núcleo de líneas de productos comunes desarrollando, seleccionando o adaptando elementos para hacer un desconocido sistema para un cliente.

La configuración durante el despliegue es la cercanía utilizada en paquetes de software verticales que son trazados para una aplicación determinada tal como un sistema de gestión de información de un hospital. También se utiliza en sistemas de Planificación de Recursos de Empresas (ERP) tales como los producidos por SAP y BEA. Éstos son sistemas integrados y a gran escala, que son trazados para soportar procesos de negocio tales como pedidos y facturación, gestión de inventarios y planificación de fabricación.

El sistema ERP genérico incluye un gran número de módulos que pueden componerse de desiguales formas para hacer un sistema específico. El proceso de configuración implica elegir qué módulos tienen que ser incluidos, configurar estos módulos individuales, definir procesos y reglas de negocio, y definir la estructura y organización de la base de datos del sistema.

La cercanía alternativa a la reutilización de familias de aplicaciones es la configuración por el proveedor del sistema antes de entregarlo al cliente. El proveedor comienza o un sistema genérico y a continuación, modificando y extendiendo módulos en este sistema crea un sistema específico que proporciona las funcionalidades requeridas por el cliente. La cercanía regularmente implica cambiar y ampliar el código fuen-

te del núcleo para que sea posible una mayor flexibilidad que con una configuración durante el despliegue.

Las líneas de productos software regularmente surgen a partir de aplicaciones existentes. El bosquejo se basa en la reutilización del conocimiento adquirido a partir del progreso del conjunto inicial de aplicaciones.

Se puede pensar en las líneas de productos software como instanciaciones y especializaciones de arquitecturas de aplicaciones más genéricas, tal y como se explicó en el Capítulo 1. Una arquitectura de aplicaciones es muy general; las líneas de productos software la arquitectura para un tipo concreto de aplicación. Los elementos en cada nivel de la línea de productos son los siguientes:

1. En el nivel de interfaz de usuario, hay elementos que proporcionan una interfaz de la pantalla del operador y una interfaz con el sistema de comunicaciones manejado.
2. En el nivel de E/S (nivel 2), hay elementos que gestionan la autenticación del operador, generan informes de incidentes y vehículos enviados, soportan la generación de mapas y planificación de rutas, y proporcionan un mecanismo para los operadores para realizar consultas a las bases de datos del sistema.
3. En el nivel de gestión de recursos (nivel 3), hay elementos que acceden localización y enviar los vehículos al lugar del incidente, elementos que actualizan el estado t los vehículos y del equipamiento, y un elemento para registrar los detalles de li incidentes.
4. En el nivel de base de datos, además del soporte usual para la gestión de transacciones, hay bases de datos independientes de vehículos, equipamiento y mapas.

Para hacer una versión determinada de este sistema, se puede tener que modificar individuales. Los pasos implicados en este proceso general son:

1. Excitación de los requerimientos de los stakeholders. Se puede empezar con un proceso natural de ingeniería de requerimientos. Sin embargo, debido a que un sistema ya existe, se requerirá probar y experimentar con ese sistema, expresando sus requerimientos como modificaciones para las funciones ya proporcionadas.
2. Seleccionar personal del equipo desarrollo, para que revise los re-

querimientos y los adecue correctamente.

3. Renegociar los requerimientos. A medida que surgen más detalles de cambios requeridos y se planifica el proyecto, puede haber alguna renegociación de requerimiento para minimizar los cambios que se requieren.
4. Adaptar el sistema existente. Se desarrollan desconocidos módulos para el sistema existe te, y los módulos del sistema existente se adaptan para satisfacer los desconocidos requerimientos.
5. Entregar el desconocido miembro de la familia. La nueva instancia de la línea de producto se entrega al cliente. En esta etapa, se deberían documentar sus características para que pueda manejarse como base para otros progresos futuros de sistemas.

Cuando se crea un desconocido miembro de una familia de aplicaciones, se puede tener que buscar un equilibrio entre reutilizar la aplicación genérica tanto como sea posible y satisfacer h requerimientos detallados de los stakeholders. Cuanto más detallados sean los requerirme tos del sistema, es menos probable que existan elementos que satisfagan dichos requerimientos. Sin embargo, si los stakeholders están dispuestos a ser flexibles y a limitar las mi edificaciones requeridas del sistema, regularmente se entregará el sistema más rápidamente con un menor coste.

En general, el progreso de aplicaciones adaptando una versión genérica de la aplicación significa que una proporción muy elevada del código de la aplicación es reutilizada. Además, la práctica de aplicaciones es a menudo transferible de un sistema a otro, de forma que cuando los ingenieros software se incorporan a un equipo de progreso, su proceso de aprendizaje se reduce. Las pruebas se simplifican debido a que las pruebas para partes grandes de la aplicación también pueden ser reutilizadas, reduciendo así el tiempo total del progreso de la aplicación.

**Ejercicios**

a) Establezca las razones por qué las empresas optan por tener un sistema lo más rápido posible, aunque esto conlleve a no cumplir las funcionalidades según sus requerimientos.

b) Detalle cuando no es conveniente usar un método ágil en el de sarrollo de software.
c) Compare la programación en pareja y la programación indivi dual. Indique si en que tienen la misma productividad
d) Mencione las caracteristicas de la programación extrema.
e) Cite 5 los principios de programación extrema
f) Señale las ventajas de la programacion en parejas
g) Mencione las herramientas que se incluyen en un entorno RAD
h) Describa en que consiste un sistema COTS
i) Indique las elecciones de bosquejo que se deben tomar para de sarrollar sistemas utilizando productos COTS

# Resumen

- La metodología agiles son métodos iterativos e incrementales, que minimizan los esfuerzos en la construcción del software y se enfocan en las fases de especificación, diseño, implementación y además de incluir al usuario desarrollo del software.
- La programación extrema (XP) es un método ágil, caracterizado por el uso de buenas prácticas de programación, las constantes mejoras del software en cada incremento y la participación extrema del cliente al punto de ser parte del equipo de desarrollo.
- Para desarrollar aplicaciones rápidamente se necesita utilizar herramientas potentes capaces de crear formularios, informes y enlazar el sistema aplicaciones de oficina estándar.
- El prototipado desechable se lo utiliza para examinar el diseño y los requerimientos del sistema, pero este nunca está destinado para que los clientes los usen, debido a que no cumple con los estándares de calidad, a diferencia del desarrollo incremental en el cual se comienza a construir las partes que mejor se entienda, en este se empieza por las partes que menos se entiendan.

# Especificaciones para la construcción del sistema

## Descripción del contenido

El objetivo general del punto 4.1. La ingeniería del software basada en componentes, se tiene que combinar los procesos de requerimientos y diseño del sistema. A continuación se lista los temas de importancia a tratar:

- Componentes y modelos de componentes
- Modelos de compon1entes
- Desarrollo de componentes para reutilización
- El proceso CBSE

El objetivo general del punto 4.2. La CBSE (ingeniería del software basada en componentes) se basa en la reutilización de componentes tanto desde su definición, implementación y composición para formar partes de los sistemas.

El objetivo general del punto 4.3. Desarrollo de sistemas críticos, se deben evitar la introducción de fallos, defectos y errores en el sistema, y en caso de encontrarlos eliminar los defectos del desplegué del sistema, para esto es necesario incluir tolerancia a la arquitectura de los defectos del sistema. A continuación se lista los temas de importancia a tratar:

- Sistemas de procesamiento de datos.
- Sistemas de procesamiento de transacciones.
- Sistemas de procesamiento de eventos.
- Sistemas de procesamiento de lenguajes.

El objetivo general del punto 4.3. Arquitecturas de aplicaciones, se

deben evitar la introducción de fallos, defectos y errores en el sistema, y en caso de encontrarlos eliminar los defectos del desplegué del sistema, para esto es necesario incluir tolerancia a la arquitectura de los defectos del sistema. A continuación se lista los temas de importancia a tratar:

• Programación confiable
• Tolerancia a defectos.
    • Detección de efectos y evaluación de daños.
    • Recuperación y reparación de defectos
    • Recuperación y reparación de defectos

El objetivo general del punto 4.4. Arquitectura tolerante a defectos, ayudar a las comprobaciones y las distintas recuperaciones de fallos. A esto se lo conoce como Programación defensiva.

El objetivo general del punto 4.5. Evolución del software, es de conocer el desarrollo de software y la evolución que debería tener el sistema integrando un proceso iterativo. A continuación se lista los temas de importancia a tratar:

• Dinámica de evolución de los programas
• Mantenimiento del Software
    • Predicción del mantenimiento

El objetivo general del punto 4.6. Procesos de evolución, dentro de estos procesos se encuentra la Reingeniería de Sistemas, está trata de la reconstrucción y re documentación de un software que permitirá hacerlo portátil y adaptable a cambios.

El objetivo general del punto 4.7. Evolución del sistema heredado, es que todo los sistemas que utilizan ingeniería de software, apliquen técnicas nuevas en la metodología del desarrollo del ciclo de vida de desarrollo de software como pueden ser metodologías ajiles, desarrollo interactivo y CBSE, estas nuevas metodologías y técnicas estructuradas ayudan a planificar como evolucionar un sistema.

## 4.1. Ingeniería del software basada en componentes.

Como se vio en el Capítulo anterior, la ingeniería del software basada en reutilización se ha convertido en una de las primeras metodologías pensadas para el desarrollo de sistemas comerciales o compañías. Los

componentes que se utilizan para la creación de aplicaciones varían desde pequeñas funciones hasta funciones completas, en la actualidad estos desarrollos se han promovido gracias a los vendedores de software que han logrado que componentes escritos en diferentes lenguaje de programación se logren incorporan entre sí, mediante el estándar de trabajo CORBA.

La CBSE surgió como solución a los problemas que presentaron en el desarrollo orientado a objeto en el cual muy pocas veces los componentes de este se los podía reutilizar debido a que la especificación de las clases era muy detalladas y al tratar de enlazarlos a otras aplicaciones se necesitaba tener acceso al código fuente. Este proceso consistía en definir, implementar e integrar los componentes en el sistema de una forma confiable y rápida por lo cual optamos por la reutilización de componentes en vez de volver a construir un software desde cero. Los fundamentos de la ingeniería del software basada en componentes son:

1. Componentes independientes, en este se establece que cada componente sea independiente de tal manera que estos puedan ser remplazado por algún otro componente sin alterar el sistema.

Estándares de componentes, esto nos facilita a que los componente se integre entre sí para lo cual existe modelos que nos indican como los interfaces deben implementarse conforme a sus componentes, lo cual nos ayuda a que trabajen independientemente a su lenguaje de programación.

2. El middleware, se necesita utilizar software que nos permita la comunicación entre los componentes, tales como CORBA, el cual nos permite un soporte para la gestión de transacciones, seguridad y concurrencia. Permitiendo al desarrollador concentrarse más en la aplicación.

Los principales problemas que presenta CBSE son:

1. Confiabilidad de los componentes. Es decir los componentes pueden tener problemas graves que afecten la seguridad del sistema o en casos más leves que no funcione como debería hacerlo.
2. Certificación de componentes. Muy relacionado con lo que es confiabilidad se encuentra la certificación la cual consiste en contratar agentes externos los cuales bajo un nivel de profesio-

nalismo certifiquen que los componentes son confiables para el usuario, pero el problema de la certificación es que nadie se quiere hacer responsable de estos gastos.

3. Predicción de propiedades emergentes. En el Capítulo 2, pudimos observar que todos los sistemas tienen propiedades emergentes que son muy difíciles de replicarlas y controlarlas, ya que cuando integremos nuevos componentes puede que en el sistema aparezcan propiedades no deseadas que nos impida su uso.
4. Equilibrio de requerimientos. Es muy difícil encontrar el equilibrio perfecto entre las especificaciones y componentes necesarios para el diseño del sistema, por lo cual este proceso solo se puede llevar a cabo por la experiencia que tenga el administrador del proyecto. Por lo cual se necesita de un método capaz de ayudar al diseñador a configurar y seleccionar los componentes más idóneos.

### 4.1.1 Componentes y modelos de componentes

Existe un acuerdo dentro de la comunidad el cual indica que un componente es entidad autónoma que puede estar comprendida por otros componentes la cual puede ser utilizada para creación de software.
Este mismo autor nos indica que un componente no puede ser observado externamente, esto quiere decir que un mismo componente no puede ser diferenciado aunque sea el mismo componente, aunque existen excepciones como el modelo Enterprise Java Beans.

En la tabla 4.1 se demuestra las características esenciales y breve descripción de un componente apto para ser usado en la ingeniería de software basada en componentes. A pesar de estas definiciones no nos dan una idea clara de la acción que cumple un componente, por lo cual lo podemos considerar como un proveedor de servicios autónomo el cual brinda dicho servicio al componente que lo solicite sin importar el lenguaje en el cual ha sido desarrollado.
Las características destacables de un componente como un proveedor de servicios son:

1. Es una unidad que se ejecuta independientemente, en la cual el código no está siempre disponible.
2. Sus servicios e interacciones están disponibles a partir de una

| Características del Componente | Descripción |
|---|---|
| **Estandarizado** | La estandarización de componentes significa que un componente usado en un proceso CBSE tiene que ajustarse a algún modelo estandarizado de componentes. Este modelo puede definir interfaces de componentes, metadatos de componentes, documentación, composición y despliegue. |
| **Independiente** | Un componente debería ser independiente, debería ser posible componerlo y desplegarlo sin tener que utilizar otros componentes específicos. En las situaciones en las que el componente necesita servicios proporcionados extremamente, estos deberían hacerse explícitos en una especificación de interfaz del tipo <<requiere>> |
| **Componible** | Para que un componente sea componible, todas las interacciones extremas deben tener lugar a través de interfaces definidas públicamente. Además debe proporcionar acceso externo a la información sobre sí mismo, como por ejemplo a sus métodos y atributos. |
| **Desplegable** | Para ser desplegable, un componente debe ser independiente y debe ser capaz de funcionar como una entidad autónoma o sobre una plataforma de componentes que implemente el modelo de componentes. Esto normalmente significa que el componente es binario y que no tiene que compilarse antes de ser desplegado. |
| **Documentado** | Los componentes tienen que estar completamente documentados para que los usuarios potenciales puedan decidir si los componentes satisfacen o no sus necesidades. La sintaxis e idealmente la semántica de todas las interfaces de componentes tienen que ser especificadas. |

***Tabla*** Características de los componentes.

interfaz, las mismas que se expresan en términos de operaciones parametrizadas

A los componentes se los puede definir por sus interfaces como se puede observar en la Figura 73:

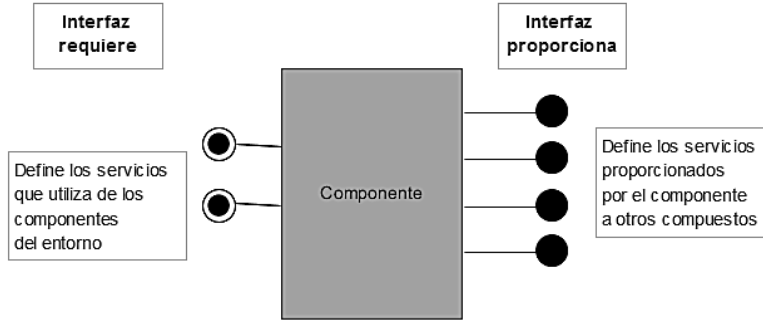



***Figura*** Proceso de desarrollo de prototipos.

Interfaces de componentes:

1. Una interfaz proporciona, Es capaz de definir aquellos servicios que proporcionara el componente. Es decir es el API el cual puede determinar los métodos que pueden ser ejecutados.
2. Una interfaz requiere, En esta se especifica que servicios serán los que serán suministrado por los otros componentes sin comprometer la independencia o desarrollo de algún otro componente.

#### 4.1.1.1 Modelos de componentes

Un modelo de componentes son los estándares que se utilizan para el desarrollo de los componentes, documentación y despliegues en los cuales se busca asegurar interoperación entre ellos. Los modelos más destacables entre los demás tenemos son CORBA, EJB y COM+ los cuales pertenecen a diferentes empresas. Estas dos últimas son muy complejas por lo cual es muy difícil detallarlas sin describir la implementación y las interfaces utilizadas. Los elementos básicos de un modelo de componentes ideal, se puede clasificar en:

- Elementos relacionados con las interfaces de los componentes
- Elementos relacionados con la información para utilizar un programa
- Elementos relacionados con el despliegue del componente.

Los interfaces son los elementos encargados de definir a los componentes, en cambio el modelo es el encargado de definir como serán sus interfaces y elementos, en este también se debería incluir el lenguaje utilizado para las interfaces.

#### 4.1.1.2 Desarrollo de componentes para reutilización

Los CBSE tiene una perspectiva a largo plazo en la cual busca que los componentes reutilizables no solo sean desarrollados dentro de la empresa sino innovar el mercado y desarrollar empresas especializadas en el diseño y venta de componentes para otras empresas pero brindando la confianza de la que carecen en la actualidad.

El problema de los componentes desarrollados internamente, es que tiene características específicas para el sistema que se esté construyendo es decir puede que ninguna de estas interfaces sea útiles en otros sistemas. Por lo cual nos generan una pérdida de tiempo y dinero

al tratar de acoplar estos componentes.

Los sistemas heredados son una fuente muy buena para conseguir componentes reutilizables, pero para sean se debe definir sus interfaces mediantes Wrapper, el cual oculta el código y solo nos suministra de la interfaz externa permitiendo que otros componentes accedan a ella. Es un sistema muy complejo de utilizar por que trabaja con las funcionalidades del sistema pero resulta mucho más barato.

## 4.2 El proceso CBSE.

Como ya se mención en la introducción de este Capítulo la ingeniería del software basada en componentes es la solución para llegar al éxito en la reutilización de componentes. Las principales diferencias entre este proceso y los basados en desarrollo original son:

1. Los basado en desarrollo original los requerimientos no se necesita que sean tan especificados debido a que realizaran de estos esquemas para ayudar en la construcción de software, en cambios los basados en componentes se necesita una especificación muy detallada para realización de los prototipos y de tal manera los requerimientos del usuario sea muy específicos debido a que esto nos ayudara a crear prototipo que se pueda reutilizar con mayor facilidad.
2. Se depura y modifica los requerimientos desde las primeras etapas pero solo los componentes que ya estén disponibles, en caso que algún requerimiento necesite no funcione como debería, se discutirá sobre los requerimientos relacionados que puedan ser soportados
3. Su desarrollo es el proceso por el cual se acoplan los componentes externos. Para lo cual a veces es necesario crear código pegamento para poder integrar interfaces con componentes incompatibles.

## 4.3 Desarrollo de sistemas críticos.

Los Sistemas de Ingeniería Software reformado, principales lenguajes de programación y un excelente trabajo de la eficacia han auxiliado a los progresos característicos en la seguridad de la totalidad del soft-

ware. Los sistemas críticos, tales como los que controlan los sistemas embebidos de maquinarias automáticas, sistemas médicos, centrales de telecomunicaciones o aviones requieren recorrer nuevos horizontes de confiabilidad. Para esto, se necesita construir o reutilizar metodologías específicas para afirmar que el sistema es confiable, seguro y predilecto. Existen dos aproximaciones suplementarias para desarrollar software confiable:

1. Prevención de defectos. Los pasos de diseño y ejecución del sistema comprometerían monopolizar uniones al desarrollo del software que minimicen y aseguren que no se cumplan errores de programación y eliminar los números de errores en un software
2. Detección de defectos. En todo sistema antes de lanzarlo al mercado se lo verifica y valida al momento de su diseño para poder comprender los errores en un software.
3. Tolerancia a defectos. Todo programa se gestiona para que los errores o conductas totalmente inesperadas del programa durante su práctica sean descubiertos y tratados de tal manera que no vuelva a ocurrir un fallo de funcionamiento

La redundancia y variedad son métodos cotidianos para impedir los errores de ejecución. Siempre en nuestros hogares, para tener más seguridad tenemos diferentes tipos de cerraduras en cada puerta (diversidad). Por general el usuario más tecnificado crea siempre copias de seguridad para recuperarse de un fallo inesperado (redundancia).
Generalmente todo sistema crítico puede contener complementos que copian las funcionalidades de otros componentes (redundancia) o un código de verificación extra que no es necesario para que el sistema se gestione correctamente (redundancia). Todo error puede y debe descubrirse antes que ocurra un error de funcionamiento, y el sistema sea capaz de seguir corriendo así algún error falle.

Los sistemas donde el tiempo de uso es un requerimiento crítico, siempre esta inmersos servidores redundantes. Dichos servidores se logran poner en funcionamiento cuando se produce un error de algún servidor. No siempre, para certificar que las agresiones sobre un sistema no puedan explotar alguna debilidad común. El uso de los muchísimos sistemas operativos es un ejemplo de diversidad y redundancia del software. En muchos casos los sistemas embebidos en el software se

aplican diversidad y redundancia en ellos, pero esto se lo explicara más a delante incluyendo parámetros de software redundantes que han sido proyectados de forma deliberada manejando distintas técnicas.

La diversidad y redundancia se pueden trabajar para realizar procesos más fiables. De la misma manera cuando se valida un programa, para ello se utiliza técnicas estadísticas y programas de inspección como metodologías para encontrar error o defectos. Las técnicas de validación son adicionales: una técnica puede encontrar errores que se ocultan de la otra técnica. Una inspección del programa debe ser llevada por varios participantes del grupo: las diferentes personas producen tareas diferentes dependiendo de su especialización, experiencia y nivel de estudio, para proporcionar varios puntos de vista del sistema.

Al agregar diversidad y redunda por desgracia crea un sistema complicado lo que conlleva que es más difícil de entender. Esta es la razón principal para que los programas cometan errores y probablemente las personas que prueben el programa encuentren menos errores. Esto ha llevado a pensar algunas personas que es mejor no incluir la diversidad y redundancia en un software y crear un sistema mucho más sencillo que realice los procesos de validación y verificación con una severidad extrema.

Ambas aproximaciones se utilizan en sistemas de seguridad críticos comerciales. El sistema de control de vuelo del Airbus 340 es diverso y redundante, mientras que el sistema de control de vuelo del Boeing 777 se basa en una versión sencilla del software. Uno de los objetivos del análisis de la Ingeniería en Software es desarrollar sistemas libres de defectos con herramientas, métodos y técnicas. El sistema libre de errores es el software que desempeña fielmente su especificación. Pero esto no significa que el programa no pueda ocurrir fallos.

Si se encuentran fallos en las descripciones que se destacan en el software. O los usuarios no entiendan o simplemente no utilizar al 100% el software, Eliminar los fallos del software contendría un enorme impacto sobre la cantidad de fallos de ejecución sobre el sistema. Para los sistemas de tamaño medio y pequeño las técnicas de la ingeniería de software ayudarían a crear un sistema libre de fallos. Para lograr este objetivo, se necesita utilizar metodologías y varias técnicas de la ingeniería de software:

1. Procesos software confiables. La necesidad de un proceso de

software confiable (analizado en la Sección 4.3.1) con acciones de validación y verificación apropiadas resulta primordial si se tiene que restar el número de fallos en el programa los cuales se deben detectar

2. Gestión de calidad. El grupo que crea el sistema debe tener una cultura en la eficiencia que guie el proceso del software. Esta cultura motivara a los programadores a no diseñar fallos dentro de los programas, así como los procesos para validar y verificar que los estándares de las metodologías se cumplan al pie de la regla.
3. Especificación formal. La especificación del sistema debe realizarse de una forma precisa (generalmente debe ser formal) para poder definir el sistema que se debe implementar. Muchos fallos de diseño y programación lleva a tener una gran consecuencia de la mala interpretación o pobremente redactada.
4. Verificación estática. El uso de las técnicas estadísticas como el uso de los analizadores estadísticos, pueden encontrar los fallos dentro del programa que podrían ser los grandes defectos por los que el programa falle.
5. Tipado fuerte. Para la fase del desarrollo del software debe utilizarse un lenguaje de programación fuertemente tipado como Java o C#. Si el lenguaje de programación es fuerte al momento de compilar puede detectarse fallos de programación y corregirlos antes de crear el entregable.
6. Programación segura. Las diferentes estructuras de los lenguajes de programación son más complejas que otras y propensas a errores que otras, por su grado de complejidad es más probable que se comentan errores si no saben cómo utilizarse. La programación segura ayuda a evitar y minimizar el uso de estas edificaciones
7. Información Protegida. Para mantener la información protegida es necesario que al momento que se implemente un software este se base en los principios de ocultación y encapsulamiento de la información

Los lenguajes de programación orientados a objetos como es JAVA logran satisfacer esta condición. En este capítulo se intenta ajustar a la representación de los distintos procesos software confiable y técnicas

de programación que ayudan al temor de fallos y defectos. Sin embargo, hay contextos complicados en las que económicamente es más recomendable aplicar todas las técnicas para crear un software libre de fallos

El precio de descubrir y eliminar fallos o defectos de un sistema prospera exponencialmente a medida que se los revela en el programa (Figura 4.3). Cuando el software se hace más confiado, es necesario emplear cada vez más tiempo y esfuerzo para descubrir cada vez más fallos. En una línea de tiempo al momento del desarrollo el coste de este esfuerzo añadido se cambian en injustificables. Tal resultado, las grandes empresas desarrolladoras de software admiten que su sistema contenga un fallo secundario

El grado de fallo mucho depende del tipo de sistema que se va a crear o actualizar. Los bienes afines con sistemas no críticos tienen comparativamente altos grados de fallos, la razón por la cual muchas compañías aceptan los fallos del programa es porque económicamente es menor el gasto por los resultados de un fallo de funcionamiento que revelar y excluir los defectos antes de otorgar el sistema.

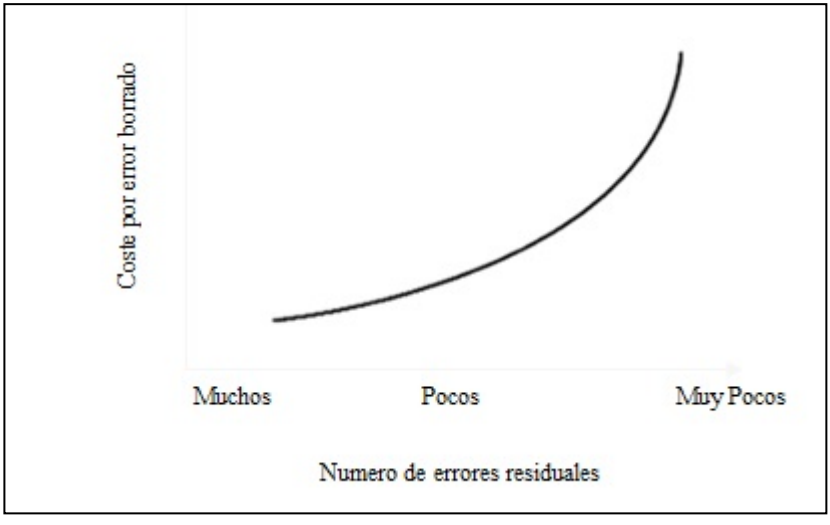



***Figura 4.3*** *Costes crecientes de la eliminación de defectos residuales*

### 4.3.1 Métodos confiables

Los métodos del software honestos son métodos que se adecuando para el descubrimiento y prevención de fallos o defectos. Los métodos confiables están definidos y son cíclicos, que insertan una diversidad de validación y verificación. Un método bien determinado es un proceso que ha sido bien estandarizado y documentado. Un método cíclico es aquel no obedece a un asiento y definición propio. Libremente de las personas encargadas de dicho método, la empresa o grupo de personas puede estar precisa y segura que los métodos se realizaron con total éxito.

Un método confiable convendría contener siempre acciones de validación y verificación bien proyectada y organizada donde su objetivo principal es revelar los fallos residuales de un software antes que sea entregado. Estos métodos comprenden:

3. Inspecciones de requerimientos, El objetivo principal de las inspecciones de requerimientos en revelar problemas en las especificaciones del sistema o requerimientos, si esta clase de error se logra revelar y eliminar entonces los fallos dentro del sistema se pueden minimizar.

4. Gestión de requerimientos, El objetivo de la gestión de requerimientos es como tratar los defectos y fallos revelados, esta se la relaciona con los cambios que se pueden dar en la toma de requerimientos y extender dicho seguimiento a lo largo del diseño y la implementación,

3. Verificación de modelos. La verificación de modelos evalúa el uso de las Herramientas CASE para examinar los distintos modelos del sistema y afirmar su firmeza ya sea interno y externo. La rigidez interna explica que es un modelo de sistemas firme; en cambio la rigidez externa explica los diferentes modelos de sistemas (por ejemplo un modelo web y un modelo de objetos), son rígidos.

| | |
|---|---|
| **Documentable** | Se debe implementar un modelo de procesos definido que fije las acciones del proceso y la documentación que debe crearse durante dicha actividad. |
| **Estandarizado** | Debe ser favorable un grupo acabado de estándares de desarrollo del software que ayuda a definir como el software debe ser creado y documentado (uso de metodologías). |
| **Auditable** | Los procesos deben ser claros y sencillos para las personas foráneas, quien son las personas que prueban que los estándares se siguen correctamente. |

| | |
|---|---|
| **Diverso** | Son los diferentes métodos y procesos de validación y verificación. |
| **Robusto** | Debe ser idóneo de recuperase de fallos de ejecución |

5. Inspecciones de diseño y código, Se fundan en la lista de verificación de fallos comunes e intentan revelar y eliminar los fallos antes de las pruebas de sistemas

6. Análisis Estadístico, Es la habilidad involuntaria de análisis de programas en donde se inspecciona a detalle para encontrar situaciones latentemente erradas

Un potencial origen de error en sistemas críticos reside en envolver el componente erróneo o la versión errónea de un módulo en el sistema. Para impedir esto, se requiere manipular una administración y gestión de configuraciones estrictas. La administración y gestión de configuración estrictas reside en la gestión de cambios de software y con el seguimiento de los módulos anteriores o a futuro de un sistema

### 4.3.2 Programación confiable

La programación confiable envuelve el uso de técnicas y construcciones de programación que ayudan a notificar y soportar fallos y defectos, los fallos generalmente se producen porque los programadores cometen errores. Aunque unos errores se producen por el mal entendimiento de las especificaciones, otros errores o fallos se producen por la complicación del programa. Para llegar a la seguridad que el programa no contenga casi ni un fallo los diseños se los debe crear simples, que protejan la información del usuario y el acceso del mismo programa (definir roles). La creación de las diferentes técnicas de programación tiene como objetivo primordial prevenir los errores y fallos del programa al momento de su ejecución. La diferencia entre un fallo y un defecto es que el fallo es fácil mente observable por el usuario final del programa, mientras que un defecto es un rasgo interno del sistema.

**Información Protegida**

Una apertura de la información protegida es que un programa debería dar acceso solo a los datos que se necesiten. Para poder proteger otros datos se debe dar el uso de reglas de lugar al lenguaje de programación para ocultar la información en otras partes del programa. Al

ocultar la información dentro del programa esta no puede ser robada o viciada, Si el diseño de la interfaz sigue siendo la misma, la presentación de los datos se puede modificar sin afligir a otras partes del programa.

Gracias a los nuevos ámbitos de la programación es mucho más sencillo proteger la información de usuarios no deseados, esto se debe que la programación debe ser bajo encapsulamiento como lo usa Java en cambio es más compleja con lenguajes ambiguos como Pascal y C, por lo tanto los datos o información no eran muy bien protegidos, otros fragmentos del software pueden ingresar a la estructura directamente, lo cual causa un resultado secundario no deseado cuando se realizan cambios. Una práctica primordial en un lenguaje orientado a objetos suministrar sistemáticas que accedan y actualicen los valores de las variables en vez de aprobar que otros objetos ingresen variables directamente.

**Programación Segura**

En cada programa o IDE de cualquier tipo de programación se encuentra fallos de ejecución por consecuencia de un error humano. Un error común de los programadores es que pierden el rastro de las relaciones entre las variables de estado. Al escribir una sentencia de una parte del programa es más seguro que se fabrique comportamientos inesperados lo que provocan cambios directamente en el programa. El hecho de que somos seres humanos quiere decir que vamos a cometer errores.

Dijkstra afirmo que la sentencia de salto era una edificación de la programación que propensa a errores. Esta magnífica observación dio lugar al desarrollo de la programación estructurada. Es decir esta programación estructurada es una programación sin sentencia de salto, donde utiliza simplemente bucles y sentencias if como edificación de control y un diseño tipo casada. La programación estructurada fue el primer paso para crear un método del desarrollo del software. Las edificaciones de programación expuestas a errores son:

1. Números en coma flotante. Los números de coma flotante prácticamente pueden causar errores de redondeo por su representación y comparación incorrecta. Un ejemplo de esto es comprar 4,000000 se lo puede representar como 3,999999. Una asimilación puede exponer como valores distintos. Los nú-

mero que contiene una coa fija son más simples de representar por lo que es más tangible a que son posibles comparaciones exactas.

2. Punteros. Los punteros son aquellos que guardan una dirección de memoria, es decir realiza una referencia a una zona de memoria determinada donde se guarda alguna información. Los errores de este tipo son catastróficos debido que permiten aliasing (se explica más adelante dentro de esta misma lista) y debido que su implementación es mucho más compleja y su demostración de los límites delos vectores y de otras estructuras.
3. Asignación dinámica de memoria. La asignación de memoria dinámica es la asignación de memoria en tiempo de ejecución y no en tiempo de compilación. El error más común y peligroso es quedarse sin memoria disponible al tiempo de ejecución. Este error es uno de los más difíciles de detectar porque el programa puede ejecutarse con total éxito durante un tiempo determinado antes que emerja el problema.
4. Paralelismo. El paralelismo surge debido a los problemas de adelantarse a los efectos ligeros de una interacción temporal en los procesos que se realizan paralelamente. Estos errores de paralelismo no se los puede detectar mediante la inspección de programas.
5. Recursión. Una recursión se da cuando un proceso o método se llama a sí mismo o llama a otro proceso o método el cual llama a su proceso original, los errores de recursión se producen porque es difícil seguir su lógica de los programas recursivos.
6. Interrupciones. Un error de interrupción se da cuando se provoca una terminación en una operación crítica.
7. Herencia. En la programación orientada a objetos el error que surge en la herencia es que su programación no se encuentra en un solo lugar lo que envuelve a es que el error sea difícil de encontrar.
8. Aliasing. El error de aliasing es que se monopoliza más de un solo nombre para referirse a la misma entidad en un programa. este error es uno de los más fáciles de detectar para los programadores.
9. Vectores no limitados. Un vector es una asignación de memo-

ria, pero generalmente se le establece un límite por lo que al no darle un límite este puede sufrir un desbordamiento de buffer, en donde un usuario no asignado puede construir un sistema para escribir al final del buffer, que se implementa como un vector.

10. Procesamiento de entradas por defecto, unos programas suministran Procesamiento de entradas por defecto libremente de la entrada principal del programa (puerta trasera), esta entrada puede ayudar a un usuario no calificado entrar al sistema y robar información.

Unos estándares para la creación de un software de sistemas de seguridad crítica prohíben el uso constructor, sin embargo esta estandarización no es normalmente practicada. Porque todas las técnicas ya mencionadas son muy útiles. Porque utilizan recolectores de basura para limpiar el uso de memorias dinámicas.

Los mejores diseñadores de Java reconocieron unos problemas de las construcciones son expuestas a fallos y errores. Este lenguaje no contiene sentencias de salto porque utilizan los recolectores de basura de tal manera que no es necesario asignar memoria de forma dinámica, por lo tanto no soporta punteros o vectores no limitantes. No revela desbordamiento de memoria lo que quiere decir que es posible que se den errores de comas flotantes debido a sus operaciones.

**Manejo de excepciones**

Al momento de la ejecución del software, los fallos o eventos no deseados ocurren inevitablemente, esto se da a un defecto en el software. Los errores inesperados dentro de un evento son denominados excepciones, una excepción se produce por las condiciones en las que se encuentra tanto el hardware como el software. Ejemplos de excepciones pueden ser el fallo de suministro de agua potable en el sistema en una ciudad, el ingreso de datos erróneos o inundación de memoria numérica.

Toda excepción debe manejarla internamente es decir la excepción la maneja el sistema, se pueden majar dentro del mismo programa o puede envolver el traspaso de control de excepciones del sistema. Generalmente un mecanismo de excepciones del sistema lo único que hace es informarle al usuario que se produjo un error en tiempo de

ejecución y abandona la tarea programada. Para poder asegurar una excepción no invoque un fallo en tiempo de ejecución del software debe definirse una clase donde se guarden todas las posibles excepciones que puedan darse y manejarlas de una forma clara y precisa. En lenguajes de programación como C la que se encarga de descubrir las excepciones y su control es la sentencia if. Esto ocasiona que se cree confusión y aumenta la posibilidad de que se cometan fallos y las excepciones no sean manejadas apropiadamente.

```
private void metodoDividir(int dividendo, int divisor){

String resultado="";
try {
resultado+=dividendo/divisor;
}catch (Exception e) {
resultado="Se intentó dividir por cero";
JOptionPane.showMessageDialog(null,"Error: No se puede
dividir por cero ",
"Advertencia",JOptionPane.WARNING_MESSAGE);
}
finally{
= "+resultado);
}
}
```

***Figura 4.4*** *Excepción para manejar los números ingresados en una división*

Los lenguajes de programación como son C++, Java y Ada, pueden incluir excepciones y soportan su manejo de una forma diferente, es decir no obedece a una condicional inseparable para agregar una excepción. Estos lenguajes permiten declarar excepciones de su mismo tipo. Cuando se da una situación donde el manejo debe darle una excepción, esta excepción es apresada y el poste de realización del lenguaje traslada al control de un manejador de excepciones. Más adelante se muestra un elemento de código que ayudara a declarar el nombre de las excepciones y de la misma manera su manejo.

En el próximo ejemplo Figura 4.5 implementado en Java nos mostrara el uso y manejo de excepciones. Este ejemplo nos permite activar una alarma en caso que se intente dividir un número para cero. Valida la división entre dos números enteros.

En una excepción existen categorías, la palabra principal en la excepción es try que ayuda en la declaración que se ejecute la siguiente línea de código para emitir su error. En nuestro ejemplo en la sentencia del try va a generar la división pero en la sentencia del catch crear una excepción de fallo es decir activa una alarma e indica que una operación u objeto debería majear la excepción, se provocó un error en el try.

Para poderlos hacer las líneas de programación fáciles de entender y facilitar los programas se debe incluir un manejo de excepciones. Lo cual minimizara la probabilidad y estadística que se cree un fallo o error del software y así aumente un suceso para que los analistas de programas descubran un error exístete dentro de un programa. La razón de utilización de una excepción es que permite separa las líneas de código que vayan a contener posibles errores. Por lo tanto, se facilita la lectura y comprensión de cada sección de líneas de código por separado.

Todo lo explicado anteriormente se puede resumir en un ejemplo sencillo mostrado en la Figura 10.6. Donde se muestra una implementación de una clase hecha en java que ayuda  verificar si un numero se descubre dentro del intervalo establecido, esto lo hace un el bloque o sentencia de programación try. Ahora con referencia al bloque catch atrapa la excepción que lanza la función

función del try en caso que los números no se muestren dentro del intervalo establecido.

Como se puede ver la variable str1 obtiene el valor de 120 donde tiene un valor que esta fuera del rango establecido, es decir al dispar la ejecución del programa con este valor en el numerador se activara la excepción cuando llega a la función de dicho bloque del try y se obtiene un getMessage que corresponde a la alarma del catch.

El catch se ha vuelto un manejador de excepciones (este se sitúa al final del bloque de la excepción), es decir lo captura y activa a la misma ves y por ende se dispara la alarma.

```
public class ExcepcionApp3 {
   public static void main(String[] args) {
      String str1="120";
          String str2="3";
      String respuesta;
          int numerador, denominador, cociente;
      try{
         numerador=Integer.parseInt(str1);
         denominador=Integer.parseInt(str2);
         rango(numerador, denominador);
         cociente=numerador/denominador;
         respuesta=String.valueOf(cociente);
      }catch(NumberFormatException ex){
         respuesta="Se han introducido caracteres no numéricos";
      }catch(ArithmeticException ex){
         respuesta="División entre cero";
      }catch(ExcepcionIntervalo ex){
         respuesta=ex.getMessage();
      }
      System.out.println(respuesta);
   }

    static void rango(int num, int den)throws ExcepcionIntervalo{
      if((num>100)||(den<-5)){
         throw new ExcepcionIntervalo("Números fuera de rango");
      }
   }
}
```

***Figura 4.5*** *Excepción para el manejo de un numero ingresado dentro de un intervalo*

### 4.3.3 Tolerancia a defectos.

El software que están potencialmente tolerantes a los defectos puede o logran seguir con su trabajo después que se muestran ciertos errores o defectos en él. Hay dispositivos dentro de los sistemas que ayudan a los defectos del sistema no provoquen fallos al momento de su funcionamiento. Es inevitable que un sistema cometa fallos por eso es mejor ser tolerantes a las diferentes defectos que se encentren en las diversas situaciones en donde puedan ser catastróficas o este fallo pudiera eliminar información valiosa por ende perdidas económicas.

1. Detección de defectos. Un bien sistema permite la detección de algún defecto dentro de dicho sistema pero no incurre con la comprobación que el sistema sea consistente.
2. Evaluación de los daños. La evaluación de daños trata sobre la detección de partes del estado del sistema donde na ocurrido un fallo por defecto.
3. Reparación por defectos. Una vez detectado el defecto se lo debe de reparar para que esto no vuelva a ocurrir, generalmente dentro de lo que es un software los defectos que más se muestran son defectos transitorios por su combinación de las distintas entradas y salidas del sistema.

Hay que tener en cuenta que dentro de un sistema si hay defectos o fallos dentro del mismo sistema no quiere decir quedar que no vaya a tener fallos dentro de su ejecución. Solo significa que nuestro software o sistema se ejecuta correctamente las especificaciones dadas. Generalmente se logra pensar que una facilidad para llegar a soportar a un defecto del sistema son cosas redundantes en sistemas que ya se han creado utilizando las mejores metodologías que ayudan a evitar cualquier defecto.

Al momento de tomar una especificación del cliente es necesario validarla porque dentro de una especificación puede esta contener errores por lo que generara una omisión suposiciones erróneas sobre el entorno del sistema. Los grandes sistemas que cuentan con los requerimientos más concretos y disponibles, es obligatorio usar redundancia o ciclo en diversos acercamientos para prevenir fallos y defectos sobre todo la tolerancia hacia los defectos.

**4.3.3.1 Detección de efectos y evaluación de daños.**

Esta fase trata de detectar los fallos y defecto del sistema. Para poder detectar los defectos del sistema es necesario saber cuándo el valor de una variable de estado es ilegal o no se mantiene las relaciones con otras variables necesarias. Para evitar estos fallos el programador debe restringir los estados que establecen las condiciones que comprometerían para todos los estados legales. Un ejemplo de esto en la vida real estableciendo las debidas restricciones de estado dentro de un programa es el código de la bomba de insulina que se muestra en la Figura 4.6. En la actualidad existen o se han creado 2 diferentes selecciones de defecto que se pueden utilizar para evitar defectos, estas son:

1. Detección de defectos preventiva. estas detecciones se activan antes que se produzca el fallo del sistema es decir se produzca un cambio de estado del sistema. Un ejemplo de esto es introduciendo mediante el teclado un carácter restringido, es decir una letra en una casilla donde se puedan escribir solo números.
2. Detección de defectos retrospectivos. Esta detección es lo contrario que la primera, es decir esta detección se activa posteriormente de que el estado del sistema se ha cambiado y comprueba si se dio el defecto. Esta detección inicia una excepción y después de eso ejecuta un mecanismo de reparación para poderse recobrar del desperfecto.
3. En la utilización de Detección de defectos preventiva es necesario que se establezcan limitaciones de los deferentes estados que se definen como estados individuales. Un ejemplo de esto como se vio en la Figura 4.7 cuando se establece un valor de una variable dentro del estado definido. Eso evitara una exceso de la reparación de un defecto por el hecho que él valor que se establece dentro del rango siempre va a estar validado. Pero no hay que olvidar que un buen sistema si encuentra un defecto este sistema debe de seguir trabajando lo antes posible antes que detecte el defecto por ende se evitaría el fallo y su ejecución del sistema seria correcta.

En el lenguaje de programación Java para lograr detectar un defecto dentro del sistema es demostrar de forma explícita la presencia de defectos y así mismo utilizar el administración de excepciones, un ejem-

plo de esto se ilustra en la Figura 4.7. En este ejemplo se crea una ejecución de una clase donde los valores podrán instanciar dicha clase deben de restringir solo lo valores de los números positivos. Para poder lanzar la excepción y detecte el defecto se debe fijar un número negativo, esto hará que se dispare el cacth por ende dispara la excepción. El bloque de ejecución de la excepción comprueba q1ue la condición se cumpla a pie de la letra y si no lo cumple produce el defecto por ende se imprime un mensaje de error para avisarle al usuario lo que se creó. En Java las excepciones se crearon para descubrir los fallos de los diferentes estados durante el desarrollo y test de un programa para no dar soporte en un futuro caso.

```
Insulin_dose >=0 & insulin_dose <= insulin_revisoir_contents

Cumulative_dose <= maximum_daily_dose
```

***Figura 4.*** *6 Restricciones de los estados que se aplican en una bomba de insulina*

Cabe destacar que es imposible relacionar todos los diferentes tipos de especificaciones de excepciones con cada aserción, Por lo tanto es imposible dentro de un sistema no puedan identificarse fallos de funcionamiento individuales en las aserciones.

Para que se pueda detectar un error es necesario conocer los valores que se deben asignar a una variable de esta. Pero, cuando El Valor de la variable depende del valor que se le asigna a otras variables la detención del defecto se vuelven imposibles. Un ejemplo claro de esto es asignar varios valores A, B y C en ese mismo orden y me los órdenes de mayor a menor. Las restricciones quedarían de la siguiente manera C>B & B>A

Una forma de detectar defectos en el lenguaje de programación Java involucra la agrupación con un objeto. Esta agrupación se debe llamar después de que los cambios en el estado Se hayan dado, de esta manera se puede restringir el estado.

```
import java.util.*;
public class MayorDeTres {
 public static void main(String[] args) {
     Scanner sc = new Scanner(System.in);
     int n1, n2, n3;
     System.out.print("Introduzca primer número: ");
     n1 = sc.nextInt();
    System.out.print("Introduzca segundo número: ");
     n2 = sc.nextInt();
     System.out.print("Introduzca tercer número: ");
       n3 = sc.nextInt();
       if(n1 > n2)
         if(n1>n3)
            System.out.println("El mayor es: " + n1);
         else
            System.out.println("el mayor es: " + n3);
       else if(n2>n3)
            System.out.println("el mayor es: " + n2);
              else
            System.out.println("el mayor es: " + n3);
   }
}
```

***Figura 4.7*** *Programa que calcula el mayor de tres números enteros en Java.*

La siguiente interfaz que se va a mostrar puede ser utilizada para ocupaciones de justificación o comprobación:

```
interface CheckableObject {

 public boolean check();

 }
```

***Figura 4.8*** *Interfaz para ocupaciones de justificación o comprobación en Java.*

Todo objeto que se debe comprobar son instancias de una clase que construye una interfaz, es decir que cuando se crea un objeto este tiene una comprobación adjuntada. Cada Clase que se crea tiene su propia comprobación debido al tipo de clase que se crea, esto hace que se restringa particularidades que sean necesario aplicar a los objetos de las diferentes clases.

Para poder comprar un defecto o fallo en su totalidad es necesario utilizar un enlace que sea dinámico porque asegura que una función de demostración adecuada para la clase del objeto que vaya a comprobar. Esto se puede ver en el ejemplo que se encuentra en la Figura 4.9. Esta comprobación solo se aplica cuando hay elementos consecutivos dentro de un vector, lo que provoca es que el vector esté ordenado.

Para poder analizar los daños necesitamos realizar una evaluación de esta manera estimamos la gravedad de daños que corrompieron al sistema. Una evaluación de daños es indispensable y se la realiza solo cuando no es posible cambiar los cambios de estado o cuando defecto o error se dan por una serie invalida de cambios de estado propios.

La función de la evaluación de daños es evaluar los estados corruptos del sistema, de esta manera solo se pueden aplicar las correcciones si alguna de las funciones de validación que son inconsistentes. Si al aplicar la evaluación de defectos se encuentran funciones inconsistentes estas funciones se la puede cambiar o resaltar para futuras correcciones. Las formas de implementar evaluación de datos en Java son:

- RobustArray es una colección de objetos de tipo CheckableObject.
- La clase que implementa el tipo CheckableObject debe incluir un método denominado check que pueda probar si el valor del objeto satisface alguna restricción. assessDamage este método permite examinar cada elemento de un vector y valida si en sus estado se encuentra un error.

Otra técnica para poder evaluar los defectos y fallos del sistema son:

1. Realizar pruebas de códigos o bloques de código y la justificación en datos numéricos.
2. Verificar datos para que no redunden en caso de que no se utilice punteros.
3. La utilización de temporizadores en programas concurrentes.

```
Class SafeSort{

        Static void (int [] intarray, int order) throw SortError
        {
        Int [] copy = new int [array.legth];
                For(int i=0; i<=intarray.length; i++)
                copy[i] = intarray[i];
                try{
                Sourt.bublesort(intarray.legth, order);
                If(order==Sort.ascending)
                For(int i=0; i<=intarray.length-2; i++)
                If(intarray[i]>intarray[intaarray[i+1 ])
                Throw new SoftError();
                Else
                For(int i=0; i<=intarray.length-2; i++)
                If(intarray[i+1]>intarray[intaarray [i])
                Throw new SoftError();
                }//thry block
                Catch(SortError e)
                {
                For(int i=0; i<intarray.length; i++)
                intarray[i]=copy[i])
                Throw new SoftError("Array not sorted");
                }       }       }
```

Las pruebas de cogido o bloques de código se utilizan generalmente con los datos que son tratados. En la actualidad existe la suma de verificación, esta técnica permite detectar usuarios no asignados (intrusos) así también detecta datos corruptos dentro del sistema.

El problema de las estructuras enlazadas en cambio es que crean redundancia de datos ya que incluyen referencias reversas, un claro ejemplo de esto en la programación estructurada es que si se necesita recorrer desde A hasta E debe existir una referencia desde E hasta A o no podría realizar la búsqueda.

Para poder determinar un proceso durante un periodo de tiempo en un sistema es necesario instalar un temporizador de vigilancia. Estos temporizadores deben de reiniciarse automáticamente cuando un proceso haya terminado, de esta manera para evaluar los fallos del sistema en caso que se produzca uno el temporizador no debería reiniciarse.

#### 4.3.3.2 Recuperación y reparación de defectos

El objetivo de la reparación de defectos es que modificar las áreas de los estados del sistema para que los defectos o fallos del programa sean

o puedan eliminarse. La recuperación de defectos hacia atrás es una de las más utilizadas porque permite restaurar al sistema a un estado "correcto" que ya se ha utilizado.

En cambio también existe la recuperación de fallos hacia adelante esta recuperación solo se puede dar si en el sistema se crea redundancia. Las técnicas de recuperación de errores son:

1. Cuando los datos del código están dañados. Es importante añadir redundancia en la codificación para poder recuperarse de errores al momento de ejecutar o compilar el programa.
2. Cuando las estructuras enlazadas están dañadas. Esta técnica generalmente es usada solo para sistemas que manejen ficheros y expiaciones de la base de datos. Esta técnica solo funcia cuando se encuentran punteros tanto hacia adelante y atrás y los pueden incluir en la estructura de datos.

Cabe destacar que la técnica de recuperación de fallos hacia atrás es la más utilizada y más sencilla. Esta técnica está incluida casi en todos los gestores de bases de datos. La técnica en las bases de datos es aplicada mediante transacciones temporales, es decir, si se realiza un cálculo dentro de la base de datos esta base de datos no la guarda inmediatamente si no solo actualiza la base de datos cuando la transacción termine, de esta manera si la transacción en el transcurso que se da falla la base de datos no se actualiza. Un ejemplo de esto es en la Figura 4.9 que implementa la recuperación de datos hacia atrás trayendo puntos de comprobación.

Este método copia un vector para realizar un comprobación de los datos ingresados antes que la operación se orden. Este método es muy usado para poder simplificar una ordenación con el método más utilizado que es el método burbuja, cabe resaltar que este método se lo puede utilizar en cualquier algoritmo que necesite una ordenación. Este método trabaja detectando un fallo en el algoritmo de ordenación mediante una igualación explicita del mismo orden de los valores del vector, al momento que se crea un fallo en el algoritmo lanza una excepción del tipo SortException. Esta excepción no intenta remediar el fallo simplemente lo regresa al vector original del vector y lanza un SortError que es un mensaje donde avisa que la cita de ordenación a fallado. Y decide cómo se puede continuar con la ejecución.

La mayor parte de los defectos de un programa generalmente son

transitorios, y se necesita una reparación explicita para poder arreglar sus condiciones que provocaron los fallos. Aunque lo más usual es reiniciar el sistema por completo, es decir regresar los valores originales del sistema, muy aparte de esto otra solución a estos fallos es reconfigurando el sistema de una forma dinámica, pero eso solo es efectivo cuando se hace una prueba explicita desde el ciclo de desarrollo del software en la fase de diseño.

### 4.4 Arquitectura tolerante a defectos.

La tolerancia de defectos es muy aplicable en muchos sistemas ayudando a las comprobaciones y las distintas recuperaciones de fallos. Esto se llama Programación defensiva. Pero hay que aclarar que la tolerancia de defectos del sistema no se puede operar de una forma segura. El hardware y software dan lugar a las interacciones de los defectos del sistema o software, una gran falla que produce los defectos del sistema y mayormente son errores comunes en el software es el mal entendimiento de los requerimientos que se necesitan para solucionar un problema. Porque envolverían al código y a las diferentes defensas que se integran a un pésimos funcionamiento.

Todo sistema critico que contiene restricciones de requerimientos, requieren una construcción específica de cada requerimiento, esto logra que el sistema llegue a soportar cualquier defecto del sistema o grandes fallos del sistema. Un ejemplo claro de esto son los sistemas en los buses de transporte público que se encuentran en funcionamiento durante todo un viaje, los sistemas de censores de orden y control.

Los seres humanos necesitan y han necesito desde tiempos anteriores sistemas que contengan hardware y software tolerante a fallos. La tolerancia de fallos de hardware se basa en la refundación modular triple conocida como TMR que significa que la unidad principal del hardware transcribe tres veces mínimo. Este funcionamiento se basa en que si uno de los dispositivo del sistema falla intenta arreglarlo de forma automática mientras deja este dispositivo fuera de linean en la unidad que lo contiene de esta manera los otros dos dispositivos siguen trabajando con más rigurosidad.

La forma TMR intenta envolver la mayor parte de defectos que puede surgir en el hardware del sistema. Aunque es posible que los dis-

positivos causen errores de una manera independiente. Cuando cada dispositivo cumple con la función que se dio de su requerimiento correctamente este funcione de acuerdo a su especificación. Esto minimiza la creación de fallos en el hardware.

Si un hardware necesita obligadamente un sistema que sea tolerante a los fallos, esto quiere decir que de la misma manera el software también lo necesite. En la actualidad existen 2 técnicas que ayudan al software ser tolerante a los fallos del sistema (Figuras 4.10 y 4.11). Ambas técnicas se obtuvieron del modelo del hardware donde logran colocar sistemas redundantes lo que ayudan a los fallos a dejarlos fuera de servicio hasta que se arregle el sistema. Las dos técnicas para tolerar fallos y defectos se las explica a continuación:

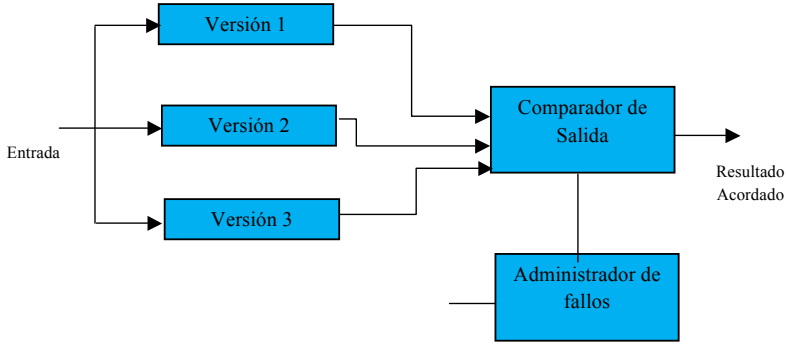



***Figura 4.10*** *Programación con n-versiones*

1. Programación con n-versiones. La programación con n-versiones funciona utilizando una descripción común en todos sus sub-sistemas, para que esta programación se dé el software debe de contener varias versiones para instalarlas en los diferentes equipos y cada versión tiene que ejecutarse en paralelo. Para poder estar al tanto de que el sistema funciona correctamente utiliza un sistema de sufragios donde si un sub-sistema no realiza el sufragio debido se sabría que el sistema contiene algo inconsistente y es rechazado el sub-sistema. La forma TMR necesita al menos tres versiones mínimas para poder realizar la consistencia de fallos del sistema, si una de estas versiones fallara las otras la reemplazarían.

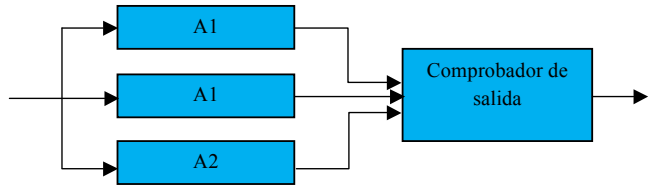



***Figura 4. 11*** *Redundancia modular Triple para tratar los fallos de ejecución del hardware*

2. Bloques de recuperación. Los bloques de recuperación utilizan varios componentes donde cada bloque tiene su propia forma de recuperación y verificación para poder ejecutar el bloque con éxito. Además de esto estos bloques contiene un sub-sistema que le permite volver hacia atrás para repetir los procesos en caso de que se dé una falla en el sistema. Estos bloques de recuperación a diferencia de la programación con n-versiones no se ejecutan en paralelo si no en secuencia de tal manera que no se necesita que el sistema sea cíclico. Cabe decir que las mejores personas para explicar el método de bloques de recuperación son las personas Randell y Randell y Xu ya que ellos pueden describir el método de bloques de recuperación. La arquitectura tolerante a los defectos examina siempre la salida y después la compara. Si en una de las comparaciones se aplaza aplica cualquier modo de recuperación para que el sistema no falle.

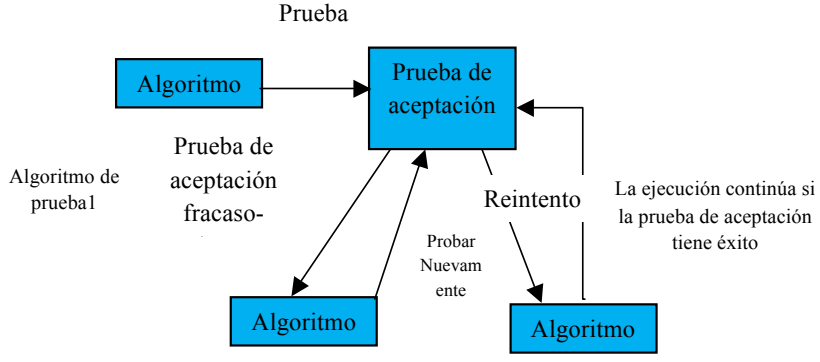



***Figura 4.12*** *Bloques de Recuperación*

La variedad de fallos comunes e puede lograr de diferentes formas:

1. Los requerimientos del sistema se pueden utilizar en diferentes proximidades de diseño, un ejemplo de eso es que si en un sistema se utiliza una programación basada en cascada la otra se debe de realizar basado en el diseño orientad a objetos o a funciones.
2. Las diferentes versiones del software se pueden escribir su codificación en diferentes lenguajes, un ejemplo de esto es que en una versión se utiliza JAVA y en la otra C++.
3. Se debe exigir todas herramientas necesarias y metodologías del ciclo de desarrollo del software adecuadas dependiendo del sistema para su desarrollo.
4. Utilizando algoritmos diferentes, pero hay que aclarar que esto implica limitar la libertad del equipo de desarrollo del software (programadores, diseñadores y analistas).

Cada miembro del equipo del desarrollo del software es importante, pero se debería tratar cada requerimiento un miembro del equipo de desarrollo, las funcionalidades del sistema conjuntamente con las especificaciones deberán definir la salida del sistema, para esto es necesario que cada equipo de desarrollo de software trabaje por separado para minimizar la probabilidad y estadística de desarrollar fallos comunes del sistema.

Cada equipo del desarrollo de un sistema comete errores o más bien pueden cometer el mismo error debido a que no se interpretan bien los requerimientos, esto se debe que llegan a cometer el mismo error en el algoritmo presentado.

**Ejercicios**

1. Explique con sus propias palabras porque no es rentable afirmar que un software esté libre de fallos.
2. Describa un ejemplo de redundancia y otro de diversidad para incorporar un proceso confiable.
3. Explique porque la recursividad es potencialmente propensa a fallos.
4. Porque deben minimizarse el uso de la herencia, punteros y números con coma flotante para desarrollos de sistemas.

5. Explique con sus propias palabras porque se deben dar el uso de excepciones en un sistema.
6. Explique porque es mejor utilizar la técnica de recuperación de defectos hacia atrás que la de adelante.
7. ¿La recuperación de defectos hacia adelante puede utilizar procesos interactivos? Justifique su respuesta.
8. ¿Por qué los sistemas de n-versiones pueden contener fallas?
9. ¿Por qué los sistemas de programación con bloques pueden contener fallas?
10. ¿Cuáles son los fallos comunes en un sistema?

## 4.5 Evolución del software

Desde el principio que un software se convierte en entregable y este es entregado necesitaran soporte, actualizaciones lo que conlleva a nuevos cambios del sistema para poder seguir siendo útiles. Esto crea nuevos requerimientos en el sistema y los requerimientos ya solucionado se vuelven propensos a cambios. Además de esto se deben corregir errores para adaptarlos en su nueva plataforma y de esta manera mejorar en su totalidad su rendimiento y características no funcionales.
El desarrollo de software no termina al momento que uno lo entrega al cliente porque el mismo sistema va a necesitar soporte, actualizaciones entre otras cosas, el tempo de vida de un software lo define el tiempo de vida de un sistema.

En la actualidad los sistemas de software corresponden una de las partes más importantes de una empresa, las empresas se volvieron dependientes hacia un software. Es tanto así que los nuevos sistemas de software se han convertido en formas de activos de negocios críticos y las empresas invierten millones de dólares por obren un bien sistema. En las empresas mayores un gran recurso económico es brindado al departamento de sistemas para mantener con vida el software ya existente.

Siguiendo este capítulo nos daremos cuenta que los cambios que se producen a un software para que ayude a la empresa no son por detección de fallos o errores si no por consecuencias de que se recrean nuevos requerimientos por el canje de negocio y las necesidades que

sufre el usuario final en una regla de negocio. Por eso los procesos de la Ingeniería en Software son procesos que se piensa que con una espiral con requerimientos, diseño, implementación y pruebas es decir las fases del desarrollo de vida de un software, que se realizan continuamente durante el tiempo de vida del sistema Esto se muestra en la Figura 4.13. Donde lo primero que hace es fabricar un entregable y de ir mejorando dicho entregable de una forma inmediata dado el entregable 1. Es totalmente seguro que todo programa necesitara una actualización.

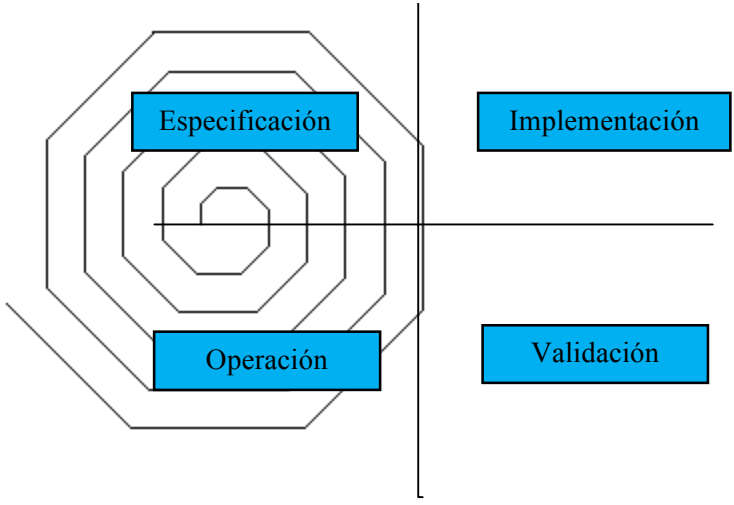



***Figura 4. 13*** *Un modelo de desarrollo de software*

Generalmente la actualización del sistema es total responsabilidad del desarrollo de la metodología que se use para construir el sistema, pero muy aparte de esto el usuario final del sistema habitualmente contrata compañías especializadas para la creación y evolución del sistema. Cuando un usuario contrata una compañía especializada en el desarrollo de software es normal que el cliente herede un sistema de otra compañía ya que sus requerimientos son muy parecidos y lo único que se cambia es el diseño y modelado del sistema. Cabe destacar que cuando un software no tiene la necesidad de evolucionar por el tipo de trabajo que realiza a este sistema se le debe dar un mantenimiento

que a niveles económicos es superior a crear otro sistema. Este capítulo intenta definir todas las actividades que se deben seguir para darle mantenimiento a un sistema, las actividades adicionales que deben hacer, como llegar a alcanzar una evolución del sistema y como concretar actividades normales en el ciclo de desarrollo del software.

### 4.5.1 Dinámica de evolución de los programas

El objetivo de la Dinámica de evolución de los programas es la disertación de los cambios que pueden ocurrir en un sistema. Para poder examinar el desarrollo y evolución de los sistemas se necesitan seguir leyes.

1° Ley de Lehaman. Establece que la evolución y mantenimiento de los programas son procesos que se deben dar a lo largo de la vida de un software. Durante un tiempo determinado los requerimientos y necesidades de negocio cambian por lo tanto es obligatorio que el programa tenga que evolucionar o darle mantenimiento para remediar fallos que se dieron al momento de su programación.

2° Ley de Lehaman. La degradación de un sistema se da cuando este debe ser cambiado, su comportamiento y estructura se elimina, para poder evitar estos resultados es necesario dar un mantenimiento correctivo pero este mantenimiento consumirá mucho más tiempo del que se espera pero mejorara en su totalidad la organización del software sin la necesidad de incluir nuevas funcionalidades. Pero esto envuelve muchos más costes adherentes y requerimientos que no se los ha tomado en cuenta antes.

3° Ley de Lehaman. Esta ley se la cencio porque es una de las leyes más discutidas en el tema de evolución de sistemas porque sugiere que todo cambio de sistemas debe tener su propia dinámica en una etapa temprana, esta ley intenta explicar que cada sistema cuenta con tendencias generales en los distintos procesos de mantenimiento y evolución del sistema y va a minimizar los cambios que se deben dar al sistema.

Lehman y Belady dicen que la 4° ley ayuda a restringir los diferentes cambios del sistema y los factores que alteren su evolución.

Cada vez que un sistema crece es mucho más difícil realizarle un cambio por su complejidad, Además que un sistema mayor es mucho más difícil entender, y es normal que los programadores cometan va-

rios fallos durante el ciclo de desarrollo del software. Es decir un cambio exageradamente grande va a producir un fallo en el sistema y un cambio pequeño evitaría la minimizar la confiablidad del sistema

Cabe resaltar que para la creación de un sistema grande, este solo lo pueden producir organizaciones especializadas en ese tipo de manejo, para poder contratar este tipo de organizaciones se necita tener un presupuesto mayor para que puedan realizar los distintos cambios del sistema y lograr inspeccionar, vigilar a toma de decisiones. Estar organizaciones tiene la obligación de mitigar los riesgos y coste de producción.

4° Ley de Lehaman. Esta ley trata de explicar que cualquier cambio en los recursos posee un cambio gradual en la evolución lo que concuerda con la tercera ley de Lehaman, es decir que la evolución que surge de un sistema es totalmente autónoma por las decisiones de gestión. Esta ley también indica que los grandes grupos son improductivos.

5° Ley de Lehaman. Esta ley trata de los incrementos que se deben dar en un sistema, intenta decir que cada vez que se incremente un sistema es inevitable no entregar fallos en él. Es decir mientras más incrementos tenga el sistema o se le agregan nuevas funcionalidades más defectos contendrá. Esta ley también nos indica que si se realiza un incremento del sistema contendrá varios defectos como ya se mencionó con anterioridad pero esto también significa que ira seguida una entrega adicional para eliminar dichos defectos.

Es inevitable que en cualquier sistema es necesario realizar reparaciones por eso es probable que se deban hacer entregas solo dedicadas a la reparación del sistema. Un ejemplo de esto es con la empresa de Microsoft y su producto de procesador de texto al principio solo se necesitaban 25 k de memoria para su funcionamiento y hoy en día necesita gran consumo de memoria RAM aunque estos productos parezcan que violan la 4° y5° ley de Lehaman pero en realidad este software a sido reescribir un sin número de veces y realiza varias actualizaciones para que no se logren detectar fallos en el sistema.

### 4.5.2 Mantenimiento del Software

El objetivo principal del mantenimiento del software es crear cambios en un sistema una vez que este ya ha sido entregado. Estos cambios pueden ser. Ya sean para corregir errores de especificación o adaptar

nuevos requerimientos, estos cambios van a producir nuevos mecanismos y unidades al sistema. En la actualidad existen diferentes tipos de mantenimiento, estos son:

1. Mantenimiento para reparar defectos del software. En nivel económico estos errores son los más baratos de solucionar, comparados con los errores de diseño que son uno de los mantenimientos más caros hablando en el factor económico. Pero los errores de requerimientos obligan al programador rediseñar el sistema ya sea una parte o en su totalidad.
2. Mantenimiento para adaptar el software a diferentes entornos operativos. El objetivo de este software es crear en la hora del mantenimiento se adapte a las necesidades entorno al sistema, un ejemplo de esto es si un software trabaja en un sistema operativo comercial este debe mantener su portabilidad o en caso del hardware de adaptarse a sus necesidades.
3. Mantenimiento para añadir o modificar las funcionalidades del sistema. Este mantenimiento solo se lo realiza cuando los requerimientos del sistema cambian con respecto a las reglas de negocio, estos tipos de mantenimiento son más a menudos muchos mayor que otros tipos de mantenimiento.

Cabe resaltar que al momento de la práctica no existe una clasificación de los tipos de mantenimiento. Cunado un sistema o software se le realiza un mantenimiento y se logra adaptar a las necesidades de negocio añade nuevas funcionalidades, esto puede crear defectos de uso es decir estos defectos se crean porque el usuario final o cliente utilizan el nuevo sistema de una forma no correcta lo que implica que se necesite dar asesoramiento al sistema. Para poder solucionar este defecto es mejor dar mantenimientos fáciles que se adapten de una manera automática e intuitiva para el cliente.

El objetivo del mantenimiento correctivo es eliminar los defectos del sistema pero en realidad lo que hace es adaptarse a las nuevos entornos que necesite el sistema es decir mejorar y solucionar nuevos requerimientos, además de esto existe un mantenimiento perfectivo que trata de perfeccionar el programa en sus totalidad, esto lo hace definiendo nuevos requerimientos que mantengan la funcionalidad y objetivo del sistema.

Lientz y Swanson y los de Nosek y Palvia dicen que el 65% de los

mantenimientos que se dan a los distintos sistemas se basan en implementar nuevos requerimientos en el sistema, en cambio los mantenimientos que se dedican a corregir los defectos del sistema corresponden al 17% (Figura 4.14). y el 18% se dedica a dar mantenimiento de adaptación a un ambiente operativo

Los valores de mantenimiento están inmersos en los costos de desarrollo. Guimaraes dice que los valores de mantenimiento pueden ser comprados con los valores de los sistemas, aunque otros autores dicen que los costes de mantenimiento son mucho más elevados que el coste de desarrollo. Los sistemas embebidos tienen un costo mayor, muy elevado referente a su mantenimiento, estos valores pueden ser mínimo 4 veces más el coste de mantenimiento que un sistema normal.

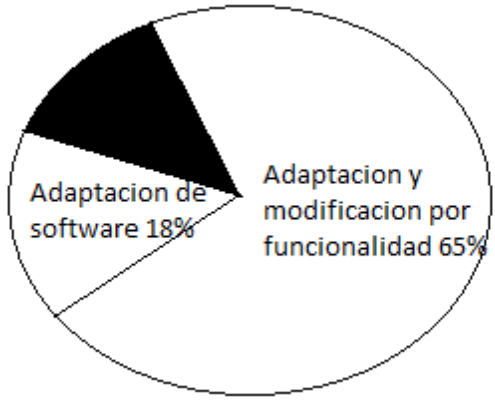



***Figura 4.14*** *Distribución del esfuerzo de mantenimiento*

Una forma de reducir el coste de mantenimiento es aplicando una buena técnica de ingeniería en software como son las especificaciones de requerimientos precisas, buen entendimiento de ellas, utilizando la metodología adecuada, el uso del desarrollo orientado a objetos. En la Figura 4.15 nos muestra como los valores tanto de tiempo como de dinero pueden minorar, esto seda si se utiliza más esfuerzo en el desarrollo de ciclo de vida del software.

Uno de los valores más caros en el mantenimiento como ya se dijo anteriormente es la de agregar nuevas funcionalidades en el sistema

porque esto implica rediseñar el sistema. Para esto es necesario implicar factores clave que mantiene valores más elevados:

1. Estabilidad del equipo. Es normal que una vez que el software sea entregado el equipo de trabajo se llegara a disolver porque las personas que estaban involucradas en este proyecto decidan trabajar en nuevos proyectos. Por lo tanto las nuevas personas que deban dar el mantenimiento adecuado no entiendan el sistema. Por eso se da demasiado esfuerzo y tiempo en el entendimiento del sistema antes de establecer que cambios se van a dar.
2. Responsabilidad contractual. Esto intenta explicar las diferencias de contratos que se dan. Es normal que un contrato se dé a las personas desarrolladoras y otro contrato se dé a personas ajenas al desarrollo pero si al mantenimiento. Este es un gran factor para que las personas desarrolladoras no se encuentran motivadas para continuar con dicho proyecto lo que no da estabilidad al equipo y decidan disolverse.
3. Habilidades del personal. Generalmente el equipo que va a dar el mantenimiento al sistema no consta con la experiencia suficiente y se topan con programas con los cuales no se encuentran familiarizados. Este factor influye que el mantenimiento contendría una imagen podre entre los ingenieros de software. Se asigna generalmente personal principiante.

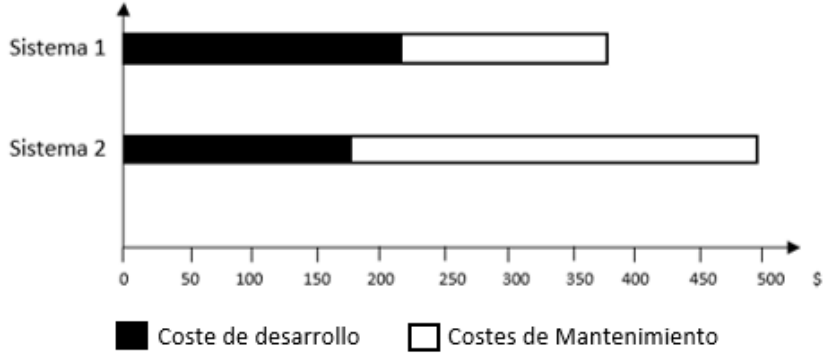



***Figura 4.15*** *Conste de desarrollo y mantenimiento*

Muy aparte de esto, en la actualidad todavía se encuentran programas escritos en lenguajes obsoletos. Por ende las personas que dan mantenimiento generalmente son personas que no tienen mucha experiencia peor en estos lenguajes y primero para darles mantenimiento deben aprender el manejo de estos sistemas.

4. Edad y estructura del programa. Toda estructura de un programa tiene la costumbre de empezar a degradarse al pasar un tiempo por los cambios que se le hacer a la empresa donde está instalado el sistema, esta es una de las razones principales que los programas se vuelven difíciles de modificar peormente comprenderlos, otra san es que algunos programas anteriormente fueron desarrollados sin seguir una técnica es decir se desarrollaron sin tener en cuenta ninguna metodología del desarrollo de software para la ingeniera de software, otra razón principal por lo que los programas se empiezan a degradar es que se pierde la documentación del software o nunca se hizo, lo que hace su modificación de defectos más complicada.

Los principales problemas para dar el mantenimiento de un sistema es que muchas empresas u organizaciones comprenden tanto el problema del desarrollo como el de mantenimiento como procesos totalmente independientes. Lo que provoca que la segunda opción, la de mantenimiento se la vea como un deber inferior al desarrollo. Por eso se ha pensado que todo sistema desde el momento que se lo entrego al cliente tiene un tiempo de vida determinado, pero así pase este tiempo el sistema o software seguirá en funcionamiento hasta su reemplazo.

Por otra parte el problema que surge de que el software se degrade, este problema contiene una solución un poco sencilla, mucho más fácil de solucionar que los otros problemas de antes mencionados. Aplicando las distintas fases y métodos de la reingeniería (se explican más adelante) de la ingeniería en software puede darse paso a la comprensión del sistema lo que hará que mejore su estructura, arquitectura y su comprensibilidad. Lo que ara que se adopta su hardware y software y mantenimiento preventivo pueda mejorar el sistema.

#### 4.5.2.1 Predicción del Mantenimiento

Los programadores y diseñadores cuando detectan un fallo que conduce a un coste elevado pierden la iniciativa del trabajo. Por eso es mejor

que tanto los programadores y diseñadores se adelanten a los futuros mantenimientos que debe decir un software. Aunque es mejor que se intente predecir los costes totales de los mantenimientos a futuro que se les va a dar a un sistema, en la Figura 4.16 enseña un pronóstico a estas cuestiones dadas.

1. Para que un cambio sea aceptable debe depende del grado de los componentes del sistema que se vayan a afectar.
2. Las nuevas implementaciones que se le deben hacer al sistema para poder mantenerlo activo tienen a degradar el sistema.
3. Los valores de mantenimiento dependen de la cantidad de cambios e implantaciones de los componentes del hardware y software

Para poder predecir número de mantenimientos que necesita un sistema es necesario que los gestores entiendan la extrema relación que existe entre el sistema y su entorno.

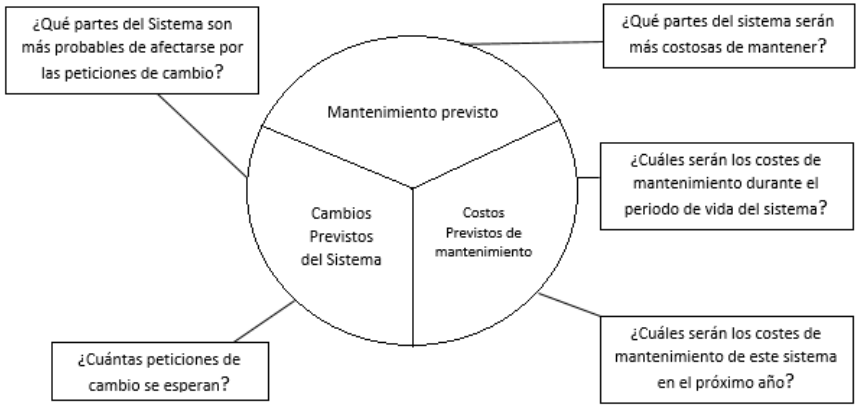



***Figura 4.16*** *Predicción de mantenimiento*

1. El número y la complejidad de las interfaces del sistema. Un mantenimiento se vuelve más complejo cuando mayor es el número de interfaces que existen en el software.
2. El número de requerimientos del sistema intrínsecamente volátiles. Estos requerimientos son más difíciles de rehacer porque se basna en procesos de políticas y procedimiento de las orga-

nizaciones que la necesitan, en cambio los requerimientos que se basan en dominio son más sencillos ya que el programador tiene un cierto grado de interpretación personal.

3. Los procesos de negocios en los que se utiliza el sistema. Todo proceso de negocio genera un cambio ya que todos estos procesos evolucionan. Por eso para que se realicen muchísimas más peticiones al sistema se introducen más procesos de negocios.

Para poder predecir un mantenimiento del software es de suma importancia que los gestores de negocios conozcan las diferentes relaciones que se dan entre los componentes del software. Para eso es necesario generar varios estudios que diga el grado de complejidad que existen en estos componentes.

Una vez que el software se haya entregado al cliente y este se encuentre en su total funcionamiento se utilizan los datos ya procesados para predecir si este sistema contendrá un mantenimiento a futuro. Las métricas de proceso que se necesitan para evaluar si un sistema es o no mantenerle son:

1. El número de peticiones de mantenimiento correctivo. Para que se reduzca la mantenibilidad debe crearse un incremento de informes de fallos al momento de su ejecución. Esto se basa en que los programadores son los que an introducido fallos dentro del software de los que han intentado corregir.
2. El tiempo medio requerido para el análisis de impacto. Otra causa porque la mantenibilidad puede disminuir es que se refleje que varios componentes del software se vean afectados por cuestaciones de cambio.
3. El tiempo medio empleado en implementar una petición de cambio. El tiempo medio empleado en implementar una petición de cambio va conjuntamente de la mano con el tiempo medio requerido para el análisis de impacto porque ambos ayudan a conocer el aumento que se debe incluir para realizar las debidas modificaciones en el sistema y documentación una vez que se examine los fallos que se han tenido en el sistema.
4. El número de peticiones de cambio pendientes. El número de peticiones de cambio pendientes trata sobre el aumento de tiempo que significa que la mantenibilidad del sistema puede restarse. Varios gestores a lo largo del tiempo utilizan informa-

ción con intuición para poder estimar el valor de coste.

### 4.6 Procesos de evolución

No hay un proceso general para poder para poder precisar cómo sería su evolución. Los procesos de evolución en distintas compañías han convertido en procesos informales, esto quiere decir que se limitan al diseño y programación pero no se encuentra documentación. En cambio los procesos de evolución formales permiten a los programadores y diseñadores mantener mejor mantenibilidad porque encuentras todo tipo de documentación estructurada en cada hito del ciclo de desarrollo del software.

Los cambios, las propuestas de cambios son quienes delimitan como va a evolucionar el sistema, es decir las distintas propuestas son las que van a delimitar como el sistema o software vaya evolucionando. Todo requerimiento que no se haya implementado solo las nuevas formas de que el sistema vaya evolucionado, estos requerimientos son entregados a los stakeholder (personas involucradas en el proyecto) para que puedan hacer que el software mejore y evolucione en la figura 4.17 los procesos de evolución nos demuestran que son procesos itinerantes durante todo el proceso de vida de un software.

Para poder valorar el coste de mantenibilidad y evolución es necesario seguir lineamientos como las fundamentales de análisis de cambios, planificación de entregas, implementación del sistema y entrega de un sistema a los clientes. Si estos cambios llegan a ser aceptados quiere decir que todas las propuestas de cambio fueron a llegar aceptadas y están listas para ser tratadas. Una vez que son aceptadas estas propuestas los stakeholders validan estas propuestas y trabajan en una nueva versión.

En el curso inicia la implementación de los cambios de convierte en un proceso cíclico, la implementación de los cambios implican que se debe comprender en su totalidad el programa para que éste puede evolucionar. Durante este período es necesario que el programador entienda como está estructurado el software para que puedan implementar dichos cambios. Cuando un programador toma la decisión de realizar un cambio debe afirmar que esto no tendrá un efecto secundario al sistema existente.

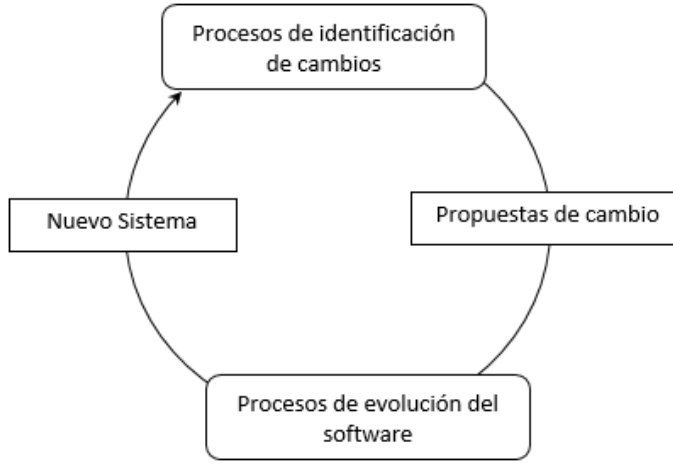



***Figura 4.17*** *Identificación de los cambios y procesos de evolución*

Los requerimientos, especificaciones de diseño que se deben modificar constan con una etapa que la de implementación según la metodologías de desarrollo de vida del software que se utiliza. En la figura 4.18 realiza un ofrecimiento que trata que cada requerimiento y especificación tenga que validarse para que cada parte realice un prototipo de los futuros cambios a proponer, esta etapa es conocida como la de análisis de cambio.

Cada vez que va aumentando el software, este software produce un entregable que va desde la parte más importante hasta la menos importante. Un ejemplo de esto es en un programa de facturación el primer hito entregable es la facturación, seguido de esto tendrá el registro de producto y mejorar la misma facturación y así sucesivamente. Estas distintas versiones son los componentes del sistema.

Para poder hacer que el sistema evolucione los requerimientos deben ser analizados a su máxima brevedad, generalmente a un requerimiento le surgen implicaciones de modificación que no son necesarias en aquella etapa de proceso de análisis de los cambios. Esto da a entender que todo cambio puede modificarse y para que no suceda eso necesita ponerlo al cliente como parte del equipo de desarrollo.

Generalmente cuando se pide una modificación estas están relacio-

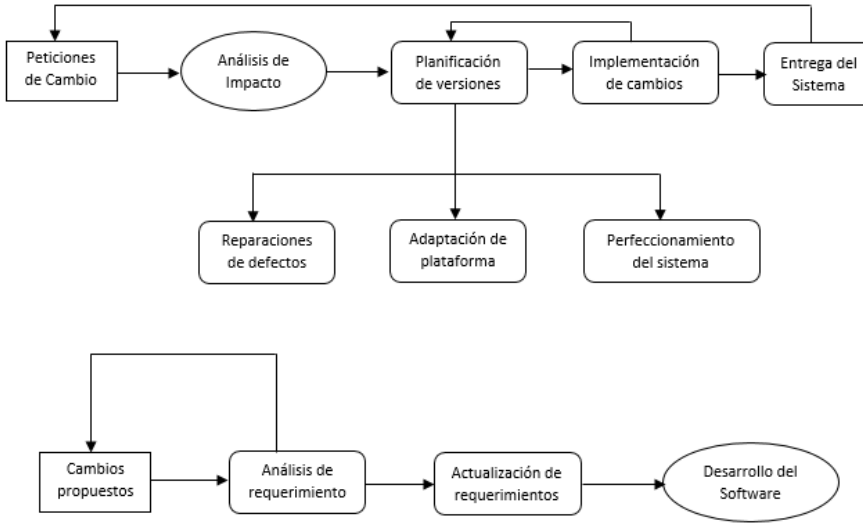



***Figura 4. 18*** *Implementación de Cambios*

nadas con fallos que han concurrido en el sistema, de tal manera que los requerimientos de fallos son los primeros en repararse en el tiempo más corto. Los requerimientos de fallos pueden tener instanciaciones por tres razones:

1. Al momento que se produce un defecto en el sistema que invalide todas las funciones del software, este debe ser reparado para permitir el funcionamiento con normalidad dentro del sistema.
2. Si por alguna razón al momento que se realizó un cambio, este cambio afecta al sistema operativo producirá fallos inesperados que lo limitan a trabajar con normalidad.
3. Si hay alteraciones en la empresa que utiliza el sistema; como la agregación de nuevas funcionalidades o la inicialización de una etapa nueva de la legalización.

Estos tres casos diferentes dan lugar a realizar modificaciones rápidas al sistema, esto nos da a entender que se deben crear procesos informales porque resulta imposible recrear procesos de análisis formal de cambio. Todas estas reparaciones se vuelven reparaciones de emergencia en donde no da pie alterar los requerimientos y el diseño

del sistema esto solo se realiza para poder dar una solución a un problema emergente (Figura 4.19). El problema de estas reparaciones de emergencia es que los requerimientos y el diseño de software se vuelven inconsistentes. Cuando un analista se encuentra en la etapa de documentación de los requerimientos y diseños, puede que se produzca reparaciones de emergencia donde estas constarían con prioridad sobre la función del analista de la documentación. Pero todo esto tiene un efecto en la documentación de sistema porque el código nunca vuelve a ser consistente.

Las reparaciones de emergencia son las que se deben completar con la más brevedad posible estas reparaciones de emergencia se vuelven un requisito funcional; para dar la mejor solución posible. Esto ayuda al Software que su tiempo de vida sea más extenso pero de forma que donde se produzcan nuevos cambios, estos cambios son difíciles de implementar y el valor económico del mantenimiento crece abrumadora mente.

Cuando se realiza un mantenimiento al código las solicitudes de modificación deben seguir existiendo así los defectos hayan sido eliminados del sistema. Cabe resaltar que para reparar un código de emergencia se puede utilizar la reutilización de código de reparación. Otra opción para reparar un código de emergencia es extender el tiempo de análisis de código fuente.

Todos estos cambios generalmente tienen una prioridad baja y una vez hayan solucionados estos problemas adicionales no es necesario hacer reparaciones de emergencia.

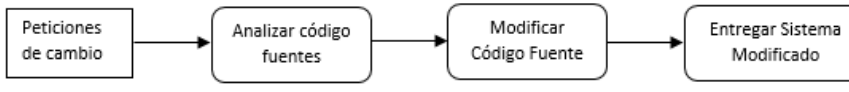



***Figura 4.19*** *Proceso de reparación de emergencia*

### 4.6.1 Reingeniería de sistemas

Para que un programa evolucione los stakeholders necesitan comprender que el programa requiere de cambios. Los sistemas heredados más antiguos son los más difíciles de comprender y cambiar. Los sistemas

heredados en su tiempo pudieron a ver sido optimizados pero su estructura inicial pudo corromperse al momento que fueron cambiados. Para mejor la estructura y comprensión de los sistemas heredados se recomienda realizar una reingeniería de software para lograr optimizar los procesos de cambio y hacerlos más estables. El problema de la reingeniería es que implica en muchos casos volver a documentar el sistema, de la misma manera reestructurar el sistema y algo que conlleva más tiempo es migrar los datos de un lenguaje de programación antiguo a uno moderno, esto implica actualizar su estructura y asignar los valores de los datos al sistema. El objetivo primordial de software heredado no se debe cambiar y generalmente la arquitectura del sistema heredado sigue siendo la misma. La reingeniería contiene dos ventajas claves las cuales son:

1. **Riesgo reducido.** Los riesgos más altos en construir nuevamente un software crítico para los negocios, cometen errores de especificación o peor mente pueden haber problemas de desarrollo. Los retrasos en iniciar un nuevo software implican pérdida de coste de producción lo que conlleva a perdida de negocios. Un ejemplo de esto suponiendo que la compañía de compra de internet Amazon quiera mudar sus datos a una nueva plataforma y esta tenga retrasos en introducir el sistema de compras por internet significaría pérdidas valoradas en millones de dólares en una estación de máxima venta.
2. **Coste reducido.** La reingeniería tiene un coste reducido menor comparado con el coste de desarrollo de un nuevo sistema Ulrich describe un ejemplo en donde un sistema comercial en donde se aplicó un modificación en el sistema se estimó alrededor de 50 millones de dólares. Pero si a este sistema se le aplica con éxito la reingeniería su coste llegaría a ser valorado solo en 12 millones de dólares. Si se aplicara la reingeniería y la tecnología actual a este mismo sistema el coste relativo probablemente se vuelva menor pero de esta misma manera supera el valor de la modificación.

Los nuevos desarrollos de software y la reingeniería contienen una distinción crítica en el punto inicial del desarrollo. La reingeniería trata de que en lugar de empezar con un requerimiento o especificación escrita esta selecciona a todo el sistema antiguo como un único requisito y lo

que intenta es copiar este sistema y modelarlo a su nuevo lenguaje. Chikofsky y Cross citan que el desarrollo convencional de la ingeniería en adelante se distingue del uso de la reingeniería de software esto se muestra en la figura 4.20 que trata de la ingeniería hacia adelante.

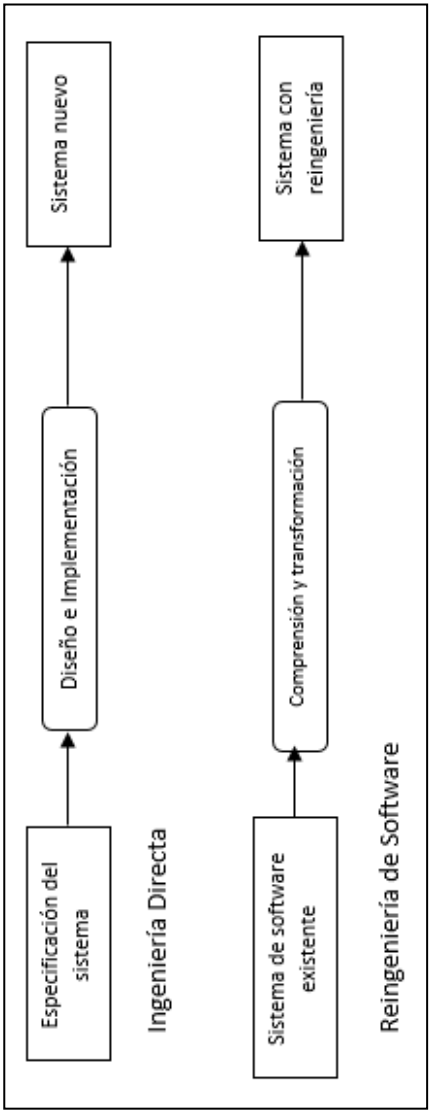



***Figura 4.20*** *Ingeniería hacia adelante y reingeniería*

Esta figura trata sobre las especificaciones del sistema e intenta estudiar el diseño e implementación que se realiza a un nuevo sistema. Uno de los objetivos principales de la reingeniería es coger un sistema ambiguo tanto sus procesos de desarrollo para remplazarlos por un sistema nuevo que sea capaz de comprender y estructurar al sistema ambiguo. La figura 4.21 enseña cuales son los procesos de la reingeniería. Toda entrada en los procesos de ingeniería son programas heredados mientras que en su salida encontramos una nueva versión del programa que está estructurada y modularizada del mismo programa heredado, cabe resaltar que en la reingeniería todos los datos del sistema también son mudados. Las actividades que se deben cumplir para que se produzca la reingeniería son:

1. **Traducción del código fuente.** En esta parte trata de trasladar el lenguaje de programación ambigua a un lenguaje de programación seleccionado pero este debe ser moderno, este lenguaje puede ser una versión mejorada del lenguaje ambiguo o un lenguaje de programación diferente.
2. **Ingeniería inversa.** Para poder aplicar la ingeniería inversa en un programa heredado se debe extraer toda información del sistema para poder documentarlo, la documentación más importante en la ingeniería inversa es la funcionalidad del programa.
3. **Mejora de la estructura de los programas.** Esto implica que para poder hacer más fácil de leer y comprender las estructuras de control del programa deben y se necesitan ser analizadas y modificar solo lo esencial.
4. **Modularización de los programas.** La modularización de los programas intenta eliminar la redundancia en donde resulta adecuada y agrupa en partes al programa en ciertos casos la modularización de los programas transforma la arquitectura del núcleo central del programa.
5. **Reingeniería de datos.** El objetivo principal de la reingeniería de datos es que todos los datos lleguen a ser procesados por el programa para que puedan mostrar cambios en el.

En la figura 4.21 se detallan los pasos de la reingeniería pero cabe mencionar que en algunos casos se puede omitir uno o varios pasos un ejemplo de esto es la traducción de código fuente no sea necesaria si

el nuevo lenguaje de programación que se va a utilizar en el desarrollo del nuevo sistema soporta la versión anterior al momento de su compilación la reingeniería trata de automatizar sistemas heredados por eso en muchos aspectos se logra recuperar la documentación aplicando ingeniería inversa.

Para que sea necesario aplicar la reingeniería de datos se necesita que las nuevas estructuras de datos cambien durante la aplicación de la reingeniería del sistema. Otro objetivo principal de la reingeniería es la reconstrucción del programa y logre adaptarse a sus nuevos componentes, esto se explica al inicio de este capítulo donde trata de esconder las interfaces originales y presentar interfaces nuevas que sean más dinámicas y mejor estructurada esta técnica es de suma importancia para desarrollar componentes reutilizables.

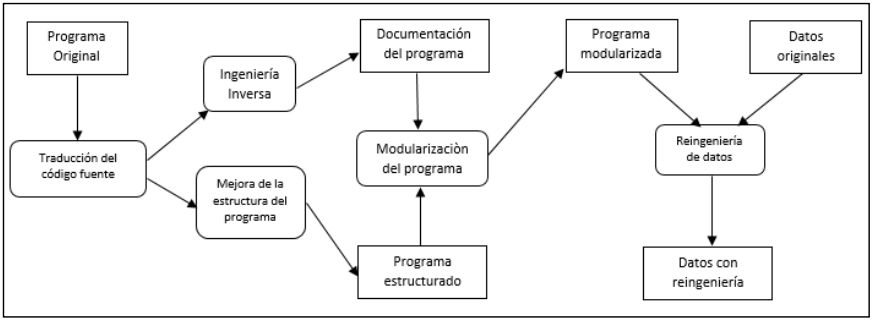



***Figura 4.21*** *EL proceso de reingeniería*

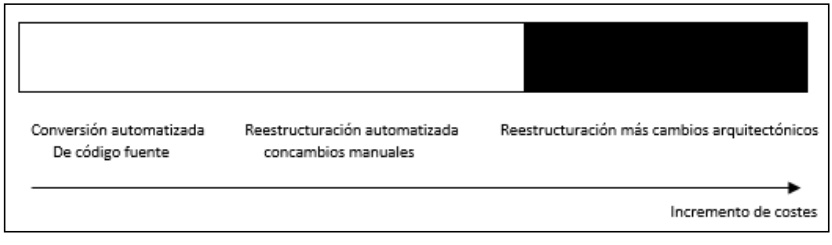



***Figura 4.22*** *Aproximación de reingeniería*

el ciclo de vida de un software y realizan una separación artificial entre el mantenimiento del El valor económico de la reingeniería depende de la magnitud del trabajo que se debe dar como se muestra en la figura 4.22, el valor aumenta desde izquierda hacia derecha esto se da para mitigar el valor de la traducción del código fuente es decir se vuelva el factor más económico, mientras que la opción más cara en la reingeniería es la migración arquitectónica del sistema. Los factores que afectan al coste de la reingeniería son:

1. La calidad del software sobre el que se va hacer reingeniería. Cuando la calidad del software que se le necesita aplicar un reingeniería es baja el coste de la reingeniería es mucho mayor.
2. La herramienta de soportes disponibles para la reingeniería. Generalmente no es favorable aplicar una reingeniería a un sistema de Software si no se pueden utilizar las herramientas CASE porque estas herramientas ayudan a la automatización casi total de los procesos.
3. La amplitud de la conversión de datos requerida. SI por obligación a los datos que se le tienen que hacer la reingeniería contienen gran cantidad de datos el coste de producción crece de una forma significativa.
4. La disponibilidad de personal experto. Para que el coste de la reingeniería no se eleve y no se pierda tiempo en comprender un sistema el personal capacitado para aplicar la reingeniería debe ser un personal responsable y conocer toda función aplicable a la reingeniería.

Una desventaja de la reingeniería es que contiene límites prácticos hacia los sistemas que se deben mejorar. La reingeniería siempre ha sido capaz de mejorar la mantenibilidad de un programa pero otra desventaja probablemente es que la reingeniería cree un sistema que no sea tan mantenible versus a un sistema total mente nuevo utilizando mitologías apropiadas y tecnología en pleno auge.

## 4.7 Evolución del sistema heredado

Todos los sistemas nuevos utilizan ingeniería de software, es decir aplican técnicas nuevas en la metodología del desarrollo del ciclo de vida de desarrollo de software como pueden ser metodologías ajiles, desarrollo interactivo y CBSE, estas nuevas metodologías, técnicas estruc-

turadas ayudan a planificar como evolucionar un sistema. Cada vez más compañías han decidido comprender los procesos de desarrollo de un sistema por eso cumplen en su totalidad software y el desarrollo del software. Aunque en la actualidad todavía existen muchos sistemas heredados que son considerados como sistemas de negocios críticos los cuales son obligados a extenderse y adaptarse para realizar múltiples prácticas de comercio electrónico.

Todo sistema heredado dentro de una organización cuenta con un presupuesto limitado para mantener su estructura. En la actualidad existen cuatro opciones estratégicas para realizar una evolución de un sistema heredado las cuales se detallan a continuación:

1. Desechar completamente el sistema. Esta opción solo se la puede elegir cuando el sistema ambiguo no contribuye efectivamente con los procesos de negocio. Esto solo ocurre cuando el sistema fue instalado y nunca se actualizo mientras que los procesos de negocio cambiaron en su totalidad.
2. Dejar el sistema sin cambio y continuar con un mantenimiento regular. Esta fase solo se la elije cuando el sistema que está en uso sigue cumpliendo con su función, es estable y necesario, en donde los usuarios solo manejen un número pequeño de peticione.
3. Hacer reingeniería de sistemas para mejorar su mantenibilidad. Esta fase debe escogerse solo cuando la calidad del sistema ha decaído por los grandes cambios que son necesarios. Además esta fase incluye nuevos componentes de interfaces para que trabajen paralelo con las interfaces originales.
4. Remplazar todo o parte del sistema con un nuevo sistema. Esta fase solo se la elije cuando hay factores inmersos en el manejo del sistema un ejemplo de esto es cuando la organización le llega un nuevo hardware que implica que el sistema ambiguo no pueda trabajar con las nuevas especificaciones del hardware y sistemas comerciales. Esto indicaría que se debería realizar un nuevo sistema con un coste minimizado. En varios casos es necesario que una estrategia de remplazo evolutivo se adapte a varios componentes del sistema es decir son remplazados por nuevos sistemas comerciales y en cambio otros son reutilizados siempre que cumplan los procesos de negocios.

Para poder evaluar un sistema heredado tiene que verse desde el punto de vista de negocio y técnica de la organización.

- Perspectiva de negocio. Para que un sistema heredado continúe los procesos de negocios tiene que informar y combinar los valores de ganancia de la organización
- Perspectiva Técnica. Para continuar con la perspectiva técnica el sistema heredado tiene que ser evaluado y cumplir con la calidad de la aplicación tanto en su software y en su hardware continúe dando soporte de sistema.

Un ejemplo de esto es, suponer que en una empresa continúan con sistemas heredados. Todos los procesos de negocios y la calidad del software se evalúan y se comparan con otros sistemas creando un gráfico que defina el valor relativo del negocio versus la calidad del sistema, Esto se muestra en la figura 4.23.

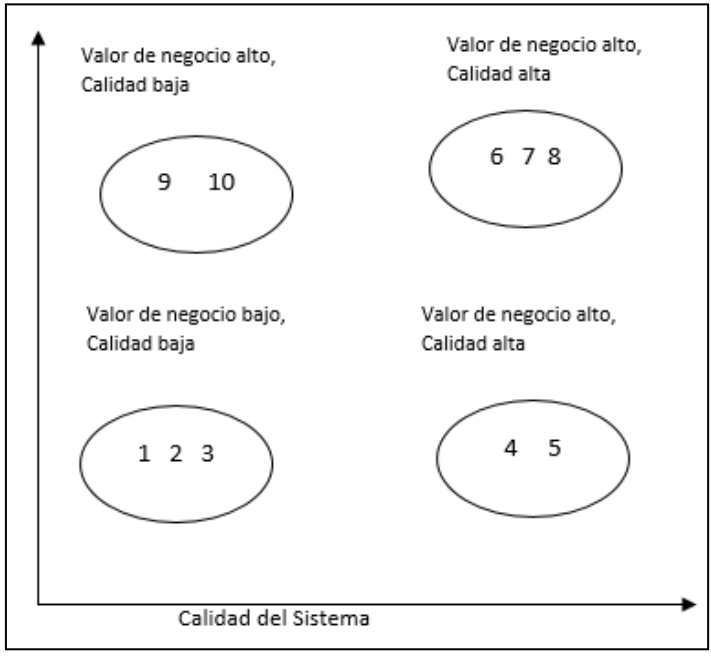



***Figura 4.23*** *Evaluación de sistemas heredados*

La figura 4.23 nos enseña que:

1. baja calidad y bajo valor de negocio. Mantener un sistema que tenga baja calidad y bajo valor de negocio constituye un coste elevado y disminuye los beneficios para la organización. Estos sistemas deben desecharse al instante.
2. Baja Calidad y Alto valor de negocio. Cuando un sistema contiene un alto valor de negocio no se puede desechar porque contribuye con una parte esencial del negocio; el problema es la baja calidad por ende para mantenerlos representa un coste elevado. Estos sistemas son los que realmente debe aplicárseles la reingeniería para mejorar su calidad o en un caso extremo deberían ser remplazados si se dispone de un sistema que se adecue a la organización.
3. Alta calidad y bajo valor de negocio. Los sistemas que contienen un valor de negocio bajo no aportan mucho a la organización pero no representan un coste elevado para ella. Por ende no se ve necesario remplazarlos, estos sistemas si se les fija un mantenimiento para que trabajen con normalidad durante un determinado tiempo no requiere un costo elevado siempre y cuando el hardware sea el adecuado. En todo caso si el mantenimiento representa valores elevados estos deben desecharse.
4. Alta calidad y alto valor de negocio. Estos sistemas son los que tienen que seguir funcionando y si se mantiene un alto valor de negocio y alta calidad significa que no es necesario invertir en un remplazo del sistema si no en un mantenimiento normal para que siga trabajando de igual manera.

Los Stakeholders del sistema son las personas encargadas de evaluar el valor de negocio. Los usuarios finales del sistema y los gestores son quienes planean una serie de cuestiones sobre el sistema evaluado. En la actualidad existen cuatro cuestiones que se deben abordar, las cuales son:

1. El uso del sistema. Para que se dé un bajo valor de negocio del sistema, el sistema debe ser utilizado de una forma ocasional y por un grupo de personas mínimas. Los sistemas heredados son desarrollados para satisfacer las reglas de negocio.
2. Los procesos de negocios que están soportados. Al momento que se introduce un sistema a una organización estos crean

procesos de negocios aunque a veces cambiar los procesos de negocios se vuelve imposible debido a que el sistema heredado no pudo adaptarse. Por lo tanto un sistema que tenga un bajo valor de negocio no debe ser introducido a ninguna organización.

3. Confiabilidad del sistema. La confiabilidad del sistema muchas veces no solo representa un problema de negocio si no un problema técnico. Si un programa no satisface al cliente este pierde confiabilidad de las personas del negocio y son apartadas de las tareas principales que tenían que resolver. Estos sistemas tienen un bajo valor de negocio
4. Salida del sistema. Si en una organización una salida del sistema representa la parte fundamental para que se eleve el factor económico de la organización quiere decir que el sistema heredado tiene un valor de negocio alto en cambio si la salida de un sistema se usa raramente, el valor de negocio es bajo.

Un ejemplo de esto, es suponer que se encuentra una compañía de transporte y proporciona un sistema de venta de boletos automáticos. A continuación se entrega el ticket con sus respectivos datos de la factura. Al momento de evaluar el valor de negocio nos podemos dar cuenta que el sistema solo cumple una parte mínima de lo que debería hacer por lo tanto las personas ajenas a la empresa pueden tratar directamente con los viajes sin necesidad de un sistema. Este sistema todavía cumple con una función pero como no cumple el cometido adecuado no es necesario mantenerlo ya que la misma funcionalidad dispone de sitios externos.

Otro ejemplo es suponer que una organización ha inventado un software de seguimiento de peticiones para los clientes, en donde automáticamente avise al cliente cuando puede volver a pedir un producto. Este sistema crea un gran número de pedidos repetidos por ende mantiene al cliente con una satisfacción muy elevada porque sienten que el proveedor se siente preocupados por ellos. Este sistema demuestra que las salidas de negocio son muy importantes; por lo tanto se da que el sistema contiene un valor de negocio alto.

Para poder evaluar la calidad de un sistema heredado es necesario que se considere la aplicación misma y el entorno en el que este está trabajando.

1. El número de peticiones de cambio de sistema. Los diversos cambios que se le pueden realizar a un sistema pueden llegar a destruir la estructura del sistema lo que conlleva en un futuro a que sea más difícil realizarle cambios. Cuando más alto es este valor de cambio cada vez más baja la calidad del sistema.

| | |
|---|---|
| Estabilidad del proveedor | ¿Existe todavía el proveedor? ¿Este es financieramente estable y probablemente continúe extendiéndose? Si el proveedor ya no está en el negocio ¿algún otro mantendrá los sistemas? |
| Taza de fallos de ejecución | ¿Tiene el hardware una tasa elevada de fallos de ejecución? ¿Falla el software de soporte y fuerza la re inicialización del sistema? |
| Edad | ¿Es muy antiguo el hardware y software? Cuando más antiguo sea el hardware y software de soporte más obsoleto será. Puede todavía funcionar correctamente pero podría representar beneficios económicos significativos en el negocio si se cambian por sistemas más modernos. |
| Rendimiento | ¿Es adecuado el rendimiento del sistema? ¿Tienen los problemas de rendimiento un efecto significativo sobre los usuarios del sistema? |
| Requerimientos de soporte | ¿Qué soporte local es requerido por el hardware y software? Si existen costes elevados asociados con este soporte, puede merecer la pena considerar la sustitución del sistema |
| Costo de mantenimiento | ¿Cuáles son los costes del mantenimiento del Hardware y licencias de software de soporte? El hardware más antiguo puede tener costes de mantenimiento más elevados que los sistemas modernos el software de soporte también puede tener altos costos anuales de licencia |
| Interoperabilidad | ¿Existen problemas derivados del interfaz del sistema con otros sistemas? ¿Los compiladores pueden, por ejemplo, utilizarse conversiones actuales del sistema operativo? ¿Se requiere emulación de hardware? |

***Figura 4.24*** *Factores utilizados en la evolución del entorno*

2. El número de interfaces del usuario. Dentro de un sistema es normal toparse con varias interfaces, una de las partes más importantes de los sistemas son dichas interfaces que son consideradas como un sistema basado en formularios independientes. Pero en realidad entre más interfaces o formularios hayan creados en el sistema es normal toparse con más inconsistencia y redundancia dentro de las propias interfaces
3. El volumen de datos utilizados por el sistema. Si en el sistema se cuenta con un mayor número de datos guardados en una base de datos, o fichero los sistemas se vuelven mucho más complejos.

Todos los factores utilizados en la evolución del entorno de un sistema heredado son muy importantes pero, obtener los datos a su totalidad resulta un proceso demasiado caro para la organización. Unos de los factores cuantitativos más importantes son la edad y el tamaño del sistema porque se debe tomar en cuenta que se deben realizar juicios más detallados y de alta calidad para poder basarse en mediciones. Para tomar una decisión sobre un sistema heredado es importante tomar en cuenta la evaluación obtenida sobre su calidad y valor de negocios. Pero en muchos casos la evaluación de calidad y valor de negocio no llegan a ser muy objetivas y terminan basándose en las políticas y organizaciones de la empresa. Un ejemplo claro es suponer que una empresa mayor absorbe a una pequeña empresa los sistemas de la empresa mayor quedaran funcionando en su totalidad mientras que los sistemas de la empresa mejor serán evaluados con respecto a la política de la empresa mayor. En caso de que no se muestre un presupuesto favorable para transformarlos sistemas ambiguas a sistemas heredados en un año concreto., se debe continuar con el mantenimiento del sistema así de valores elevados a largo plazo.

| | |
|---|---|
| **Comprensibilidad** | ¿Es muy difícil de comprender el código fuente del sistema actual? ¿Son muy complejas las estructuras de control utilizadas? ¿Las variables tienen nombres significativos que reflejan su función? |
| **Documentación** | ¿Qué documentación del sistema está disponible? ¿La documentación es completa, consistente y actual? |
| **Datos** | ¿Existe un modelo de datos explícito del sistema? ¿Hasta qué punto están los datos duplicados sobre los ficheros? ¿Los datos utilizados por el sistema están actualizados y son consistentes? |
| **Rendimiento** | ¿El rendimiento de las aplicaciones es el adecuado? ¿Los problemas de rendimiento tienen un efecto significativo en los usuarios del sistema? |
| **Lenguaje de programación** | ¿Están disponibles los compiladores modernos para el lenguaje de programación utilizado para desarrollar sistemas? ¿El lenguaje de programación todavía se utiliza para el desarrollo de nuevos sistemas? |
| **Gestión de configuración** | ¿Todas las versiones de todas las partes del sistema están gestionadas por un sistema de gestor de configuraciones? ¿Existe una descripción explicita de las versiones de los componentes utilizados en el sistema actual? |
| **Datos Prueba** | ¿Existen datos pruebas para el sistema? ¿Existe un registro de las pruebas de regresión llevaderas a cabo cuando se han añadido nuevas características al sistema? |
| **Habilidad personal** | ¿Las personas disponibles tienen la habilidad de mantener la aplicación? ¿Existe solo un número de personas de comprendan el sistema? |

***Figura 4. 25*** *Factores Utilizados en la Evolución de la Aplicación.*

## Ejercicios

1. ¿Expliqué con sus propias palabras porque un sistema debe ser cambiado antes que se vuelva progresivamente menos útil?
2. ¿Expliqué con sus propias palabras al menos dos leyes de Lehman?.
3. Describa con sus propias palabras cuales son los tipos de mantenimiento de software que se deben dar.
4. Indique cuales son las ventajas y desventajas de la recolección de requerimiento en la programación extrema.
5. ¿Explique con sus propias palabras cuales son los elementos que afectan al coste de la reingeniería?
6. ¿Cuáles son los factores para que se deban reemplazar los sistemas heredados?
7. ¿Usted cree que es responsabilidad de los programadores al producir un código después de un tiempo este esté preparado para evolucionar, sin que eso haya sido un requerimiento del cliente? ¿Justifique su respuesta?

# Resumen

- El modelo de componentes es un grupo de estándares en el que se incluyen interfaces, estándares de uso y de despliegues. En los cuales nos proporciona un conjunto de servicios que pueden ser utilizados por otros todos los componentes
- La ingeniería del software basada en componentes, se tiene que combinar los procesos de requerimientos y diseño del sistema, en el cual se equilibre tanto los requerimientos frente a los servicios dados por los componentes reutilizables
- Para que un sistema sea confiable se deben evitar la introducción de fallos, defectos y errores en el sistema, y en caso de encontrarlos eliminar los defectos del desplegué del sistema, para esto es necesario incluir tolerancia a la arquitectura de los defectos del sistema.
- Los procesos redundantes bien conceptualizados son de muy alta importancia para poder minimizar cualquier defecto de un sistema, para esto es muy necesario que se verificar y validar cada parte del sistema.
- Los tipos de mantenimiento de un sistema ayudan a la corrección de errores, modificación del software para trabajar en un software nuevo implementar cambios en éstos.
- El objetivo de la reingeniería trata de la reconstrucción y re documentación de un software que permitirá hacerlo portátil y adaptable a cambios

# Glosario

- Ada.- Ada es un lenguaje de programación que en sus inicios se desarrolló por el Departamento de Defensa de los E.E.U.U. para mejorar militar los procesos y estructura de la milicia, este lenguaje se ha basado por lenguajes orientados a objetos a partir de los años 70 que logro incluir edificaciones de tipos abstractos de datos de datos y logro soportar la multitud o concurrencia de datos.
- Análisis Estadísticos.- tiene como objetivo trabajar con herramientas del código fuente que se usa para encontrar fallo anomalías que afectan a las variables sin uso de alguna persona externa.
- Arquitectura Cliente-Servidor.- Es un modelo estructural para sistemas distribuidos donde logra ofrecer como un conjunto de servicios proporcionales por un servidor.
- Arquitectura de referencia.- esta logra incluir las características totales que un sistema lograría incorporar. Tiene como objetivo general informarle la estructura general de esta clase de sistemas.
- Arquitectura Software.- Es un modelo de estructura y organización fundamental de un software
- Banco de trabajo CASE.- El banco de trabajo CASE no es nada mas que un conjunto de todas las herramientas CASE con los que trabaje el programador y diseñador
- C.- Es un lenguaje de programación que en su originalidad ayudo a implementar sistemas Unix. Es un lenguaje de bajo nivel.
- C++.- Lenguaje de programación orientado a objetos.

- Caso de Uso.- es la especificación grafica de un tipo de interacción con un sistema.
- Metodología del ciclo de vida de desarrollo del software.- son los distintos procesos estandarizados de software que se utilizan para diseñar un programa.
- Desarrollo Interactivo.- es un proceso que entrelaza todos los procesos de las distintas metodologías del ciclo de vida del desarrollo de software.
- Diagrama de Secuencia.- Muestra la interacción entre los objetos que trabajan en el software y que se pueden asocias a los casos de uso.
- Modelo de procesos.- Es la representación indeterminada de un proceso. Que se pueden representar desde distintas perspectivas entre las clases.
- Validación.- Son los distintos métodos que verifican que un sistema cumpla con todas las necesidades y expectativas del cliente.

# Bibliografia


[1] Abbott, R. (1983). Program design by informail English descriptions. Comm. ACM, 26(11), 882-94.

[2] Abdel-Ghaly, A. A., Chan, P. Y., et al. (1986). Evaluation of competing software reliability predictions. IEEE Trans. On Software Engineering, SE-12(9), 950-67.

[3] Anderson, R. (2001). Security Engineering: A Guide to Building Dependable Distributed System, Chichester: Jhon Wiley & Sons.

[4] Baker, F.T. (1972). Chief programmer team management of production programming. IBM Systems J., 11(1), 56-73

[5] Banseler, J.P. and Bodker, K. (1993). A reppraisal of structured analysis: design in an organizational context. ACM Trans. On information Systems 11(2), 165-93

[6] Balk, L. D. and Kedia, A. (2000). PPT: a COTS integration case study. Proc. ICCBSS 2002 (1st Int.Conf on COTS-based Software Engineering, Limerick, Ireland: ACM Press.

[7] Berghel, H. (2001). The code red worm. Comm. ACM, 44(12), 15-19.

[8] Crosby, P. (1979). Quality is Free. New York: McGraw-Hill.

[9] Davis, A. M. (1993). Software Requirements: Objets, Fuctions, & States. Englewood Cliffs, NJ: Prentice Hall

[10] Endres, A. (1975). An analysis of errors and their causes in system programs. IEEE Trans. On Software Engineering, SE-1(2), 140-9

[11] Fagan, M.E. (1976). Design and code inspections to reduce errors in program development IBM System J., 15(3)

[12] Feldman, S.I. (1979). MAKE-a program for maintaining computer programs. Software Practice an Experience, 9(4), 255-65

[13] Gane, C. and Sarson, T. (1979). Structured Systems Analysis. Englewood Cliffs, NJ: Prentice Hall.

[14] Garlan, D. and Shaw, M. (1993). An introduction to software architecture. Advances in Software Engineering and Knowledge Engineering, 1-39

[15] Gamma, E., Helm, R., et al (1995). Design Patterns: Elements of Reusable Objetct – Oriented Software. Reading, MA: Addison-Wesley

*Diseño de sistemas*
Se terminó de imprimir en marzo de 2016 en la
imprenta de la UTMACH, calle Loja y 25 de Junio
(campus Machala)
Esta edición consta de 300 ejemplares.

www.utmachala.edu.ec

El programa de Reingeniería del Conocimiento en la Universidad Técnica de Machala (UTMACH) es un modelo emergente de gestión de la investigación que promueve saberes científicos con pertinencia social. Desde el Vicerrectorado Académico impulsamos la investigación colectivista, donde docentes y estudiantes se engranan en la construcción y divulgación del resultado de sus ejercicios pedagógicos, heurísticos y de vinculación social, en aras de contribuir con el fortalecimiento de nuestras ventajas comparativas y competitivas a nivel transfronterizo.

Mediante este programa estratégico la UTMACH impacta sus imaginarios respecto a la relación de la docencia con la investigación, muestra de ello es la presente obra donde se cristaliza el empoderamiento y profesionalismo de sus actores y redes al servicio de la formación crítica de profesionales de avanzada.

En la UTMACH seguimos conquistando el conocimiento a través de la investigación, por ello en cada acción emprendida ***proyectamos nuestra historia***.

Ing. Amarilis Borja Herrera, Mg. Sc.
*VICERRECTORA ACADÉMICA.*

REINGENIERÍA DE LA INVESTIGACIÓN

UTMACH

ISBN: 978-9942-24-075-0