GOBIERNO DEL ESTADO DE MÉXICO

SECRETARÍA DE EDUCACIÓN
SUBSECRETARÍA DE EDUCACIÓN MEDIA SUPERIOR Y SUPERIOR
TECNOLÓGICO DE ESTUDIOS SUPERIORES DEL ORIENTE DEL ESTADO DE MÉXICO

TECNOLÓGICO DE ESTUDIOS SUPERIORES ORIENTE DEL ESTADO DE MÉXICO

DIVISIÓN DE INGENIERÍA EN SISTEMAS COMPUTACIONALES

CUADERNILLO DE APUNTES:

DESARROLLO DE PROYECTOS DE SOFTWARE

Prof. JOSÉ ALFREDO ORTEGA HERNÁNDEZ

LA PAZ, ESTADO DE MÉXICO JULIO 2011

Tabla de contenido

Objetivos

Objetivo general

Diseñar y construirá un proyecto de software conforme a los requerimientos establecidos en el dominio del proyecto de software tomando como en cuenta los estándares actuales y las mejores prácticas en el desarrollo del mismo.

Introducción

Una de las características más destacadas de las sociedades modernas es el uso de la tecnología en la cual en muchos de los caso se involucra el desarrollo de sistemas de software que apoyan las actividades diarias en empresa, en nuestra casa, en procesos de comunicación donde interviene el manejo de información o en los sistemas de producción donde se ha sustituido la mano del hombre por maquinas que controlan la producción atreves de dispositivos programados y grabados en una placa de silicio.

El software es algo indispensable en nuestra vida diaria que de manera general se entiende como programas de computadora; el desarrollo de software es una actividad donde se involucra la ingeniería; que es el "estudio y aplicación, por especialistas, de las diversas ramas de la tecnología." [1] El desarrollo de sistemas de software no es el simple hecho de programar computadoras, instrucciones en una placa de circuitos integrados para que realice una determinada función o el realizar grandes programas que se ejecuta en una poderosa computadora que puede realizar miles de operaciones por segundo o que de igual manera se puede ejecutar en una PDA o en algún otro dispositivo electrónico; por lo que este proceso involucra a la Ingeniería de Software que comprende todos los aspectos de la producción de software desde las etapas iniciales de la especificación del sistema (Somerville, 2005, p.6).

Desarrollar un software significa construirlo simplemente mediante su descripción. Está es una muy buena razón para considerar la actividad de desarrollo de software como una ingeniería. Por lo que un producto de software no se limita solo a programas de computadora sino a la documentación asociada a este en las distintas etapas que interviene desde su concepción, análisis, diseño, implementación, pruebas y mantenimiento. El software no estaría completo si no existiera una especificación de requerimiento o un diseño de la arquitectura, crear software es muy similar a la creación y diseño de muchas otras área como la arquitectura donde podemos empezar diseñando una casa o departamento hasta un gran rascacielos o el puente más largo que comunica dos ciudades. La ingeniería de software como otras ingeniería hace uso de metodologías que involucran herramientas métodos procedimientos y técnicas para realizar un proyecto por eso es que en este manual se intanta dar una descripción de los pasos que se involucran en el desarrollo de software de acuerdo los requerimientos de los diferentes dominios de de su arquitectura.

[1] Definicion de acuerdo al diccionario de la real academia española de la lengua

1. Conceptos Introductorios.

1.1 La arquitectura “4+1”.

Todos hemos visto muchos libros y artículos donde se intenta capturar todos los detalles de la arquitectura de un sistema usando un único diagrama. Pero si miramos cuidadosamente el conjunto de cajas y flechas que muestran estos diagramas, resulta evidente que sus autores han trabajado duramente para intentar representar más de un plano que lo que realmente podría expresar la notación. ¿Es acaso que las cajas representan programas en ejecución? ¿O representan partes del código fuente? ¿O computadores físicos? ¿O acaso meras agrupaciones de funcionalidad? ¿Las flechas representan dependencias de compilación? ¿O flujo de control? Generalmente es un poco de todo.

El modelo de 4+1 vistas fue desarrollado para remediar este problema. El modelo 4+1 describe la arquitectura del software usando cinco vistas concurrentes. Tal como se muestra en la Figura 1.1, cada vista se refiere a un conjunto de intereses de diferentes stakeholders del sistema.

- La vista lógica describe el modelo de objetos del diseño cuando se usa un método de diseño orientado a objetos. Para diseñar una aplicación muy orientada a los datos, se puede usar un enfoque alternativo para desarrollar algún otro tipo de vista lógica, tal como diagramas de entidad-relación.
- La vista de procesos describe los aspectos de concurrencia y sincronización del diseño.
- La vista física describe el mapeo del software en el hardware y refleja los aspectos de distribución.
- La vista de desarrollo describe la organización estática del software en su ambiente de desarrollo.

Los diseñadores de software pueden organizar la descripción de sus decisiones de arquitectura en estas cuatro vistas, y luego ilustrarlas con un conjunto reducido de casos de uso o escenarios, los cuales constituyen la quinta vista. La arquitectura evoluciona parcialmente a partir de estos escenarios.

Arquitectura del software = {Elementos, Formas, Motivación/Restricciones}

Para cada vista definimos un conjunto de elementos (componentes, contenedores y conectores), captamos la forma y los patrones con que trabajan, y captamos la justificaci´on y las restricciones, relacionando la arquitectura con algunos de sus requisitos.

Cada vista se describe en lo que llamamos “diagrama” que usa su notación particular. Los arquitectos también pueden usar estilos de arquitectura para cada vista, y por lo tanto hacer que coexistan distintos estilos en un mismo sistema.

La arquitectura del software se trata de abstracciones, de descomposición y composición, de estilos y estética. También tiene relación con el diseño y la implementación de la estructura de alto nivel del software.

Los diseñadores construyen la arquitectura usando varios elementos arquitectónicos elegidos apropiadamente. Estos elementos satisfacen la mayor parte de los requisitos de funcionalidad y performance del sistema, así como también otros requisitos no funcionales tales como confiabilidad, escalabilidad, portabilidad y disponibilidad del sistema.

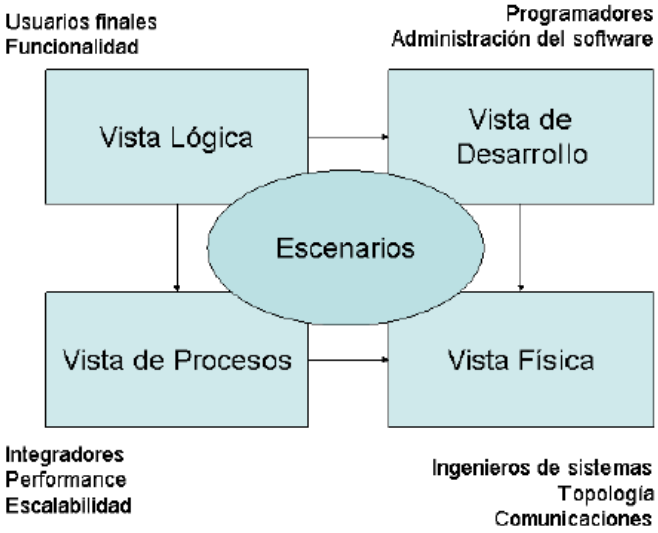


Figura 1.1: Modelo de “4+1” vistas

La Arquitectura Lógica

La arquitectura lógica apoya principalmente los requisitos funcionales –lo que el sistema debe brindar en términos de servicios a sus usuarios. El sistema se descompone en una serie de abstracciones clave, tomadas (principalmente) del dominio del problema en la forma de objetos o clases de objetos. Aquí se aplican los principios de abstracción, encapsulamiento y herencia. Esta descomposición no sólo se hace para potenciar el análisis funcional, sino también sirve para identificar mecanismos y elementos de diseño comunes a diversas partes del sistema.

La Vista de Procesos

La arquitectura de procesos toma en cuenta algunos requisitos no funcionales tales como la performance y la disponibilidad. Se enfoca en asuntos de concurrencia y distribución, integridad del sistema, de tolerancia a fallas. La vista

de procesos también específica en cuál hilo de control se ejecuta efectivamente una operación de una clase identificada en la vista lógica.

Un proceso es una agrupación de tareas que forman una unidad ejecutable. Los procesos representan el nivel al que la arquitectura de procesos puede ser controlada tácticamente (comenzar, recuperar, reconfigurar, y detener). Además, los procesos pueden replicarse para aumentar la distribución de la carga de procesamiento, o para mejorar la disponibilidad.

Partición. El software se particiona en un conjunto de tareas independientes: hilo de control separado que puede planificarse para su ejecución independiente en un nodo de procesamiento.

Podemos entonces distinguir:

- *tareas mayores* son elementos arquitectónicos que pueden ser manejados en forma univoca. Se comunican a través de un conjunto bien definido de mecanismos de comunicación inter-tarea: servicios de comunicación síncrona y asincrónica basados en mensajes, llamados a procedimientos remotos, difusión de eventos, etc. Las tareas mayores no debieran hacer suposiciones acerca de su localización con otras tareas dentro de un mismo proceso o un mismo nodo de procesamiento.
- *tareas menores* son tareas adicionales introducidas localmente por motivos de implementación tales como actividades cíclicas, almacenamiento en un buffer, time-out, etc.). Pueden implementarse en Ada por ejemplo, o como hilos de control liviano (threads). Pueden comunicarse mediante rendezvous o memoria compartida.

El flujo de mensajes y la carga de procesos pueden estimarse en base al diagrama de procesos. También es posible implementar una vista de procesos “vacía”, con cargas dummy para los procesos y medir entonces su performance en el sistema objetivo.

Vista de Desarrollo

La vista de desarrollo se centra en la organización real de los módulos de software en el ambiente de desarrollo del software. El software se empaqueta en partes pequeñas –bibliotecas de programas o subsistemas– que pueden ser desarrollados por uno o un grupo pequeño de desarrolladores. Los subsistemas se organizan en una jerarquía de *capas*, cada una de las cuales brinda una interfaz estrecha y bien definida hacia las capas superiores.

La vista de desarrolla tiene en cuenta los requisitos internos relativos a la facilidad de desarrollo, administración del software, reutilización y elementos comunes, y restricciones impuestas por las herramientas o el lenguaje de programación que se use. La vista de desarrollo apoya la asignación de requisitos y trabajo al equipo de desarrollo, y apoya la evaluación de costos, la planificación, el monitoreo de

progreso del proyecto, y también como base para analizar reusó, portabilidad y seguridad. Es la base para establecer una línea de productos.

La vista de desarrollo de un sistema se representa en diagramas de módulos o subsistemas que muestran las relaciones exporta e importa. La arquitectura de desarrollo completa sólo puede describirse completamente cuando todos los elementos del software han sido identificados. Sin embargo, es posible listar las reglas que rigen la arquitectura de desarrollo –partición, agrupamiento, visibilidad– antes de conocer todos los elementos.

Arquitectura Física

Mapeando el software al hardware

La arquitectura física toma en cuenta primeramente los requisitos no funcionales del sistema tales como la disponibilidad, confiabilidad (tolerancia a fallas), performance (rendimiento), y escalabilidad. El software ejecuta sobre una red de computadores o nodos de procesamiento (o tan solo nodos). Los variados elementos identificados –redes, procesos, tareas y objetos– requieren ser mapeados sobre los variados nodos. Esperamos que diferentes configuraciones puedan usarse: algunas para desarrollo y pruebas, otras para emplazar el sistema en varios sitios para distintos usuarios. Por lo tanto, el mapeo del software en los nodos requiere ser altamente flexible y tener un impacto mínimo sobre el código fuente en sí.

Escenarios

Todas las partes juntas

Los elementos de las cuatro vistas trabajan conjuntamente en forma natural mediante el uso de un conjunto pequeño de escenarios relevantes –instancias de casos de uso más generales– para los cuales describimos sus scripts correspondientes (secuencias de interacciones entre objetos y entre procesos. Los escenarios son de alguna manera una abstracción de los requisitos más importantes. Su diseño se expresa mediante el uso de diagramas de escenarios y diagramas de interacción de objetos.

Esta vista es redundante con las otras (y por lo tanto “+1”), pero sirve a dos propósitos principales:

- como una guía para descubrir elementos arquitectónicos durante el diseño de arquitectura tal como lo describiremos más adelante
- como un rol de validación e ilustración después de completar el diseño de arquitectura, en el papel y como punto de partido de las pruebas de un prototipo de la arquitectura.

Correspondencia entre las Vistas

Las distintas vistas no son completamente ortogonales o independientes. Los elementos de una vista están conectados a los elementos de las otras vistas siguiendo ciertas reglas y heurísticas de diseño.

De la vista lógica a la vista de procesos. Identificamos varias características importantes de las clases de la arquitectura lógica:

- Autónoma: ¿Los objetos son activos, pasivos o protegidos?
  - un objeto activo toma la iniciativa de invocar las operaciones de otros objetos o sus propias operaciones, y tiene el control completo sobre la invocación de sus operaciones por parte de otros objetos.
  - un objeto pasivo nunca invoca espontáneamente ninguna operación y no tiene ningún control sobre la invocación de sus operaciones por parte de otros objetos.
  - un objeto protegido nunca invoca espontáneamente ninguna operación pero ejecuta cierto arbitraje sobre la invocación de sus operaciones.
- Persistencia: ¿Los objetos son permanentes o temporales? ¿Qué hacen ante la falla de un proceso o un procesador?
- Subordinación: ¿La existencia o persistencia de un objeto depende de otro objeto?
- Distribución: ¿Están el estado y las operaciones de un objeto accesibles desde varios nodos de la arquitectura física, y desde varios procesos de la arquitectura de procesos?

En la vista lógica de la arquitectura consideramos que cada objeto es activo y potencialmente “concurrente”, teniendo comportamiento en paralelo con otros objetos, y no prestamos más atención al grado preciso de concurrencia que requerimos para alcanzar este efecto. Por lo tanto, la arquitectura lógica tiene en cuenta sólo el aspecto funcional de los requisitos.

Sin embargo, cuando definimos la arquitectura de procesos, implementar cada objeto con su propio thread de control (e.g., su propio proceso Unix o tarea Ada) no es muy práctico en el estado actual de la tecnología debido a la gran sobrecarga que esto impone. Más aún, si los objetos son concurrentes, debería haber alguna forma de arbitraje para invocar sus operaciones.

Por otra parte, se requieren múltiples threads de control por varias razones:

- para reaccionar rápidamente a ciertas clases de estímulos externos, incluyendo eventos relativos al tiempo
- para sacar partido de las múltiples CPUs en un nodo, o los múltiples nodos en un sistema operativo
- para aumentar la utilización de la CPU, asignando la CPU otras actividades mientras algún thread de control está suspendido esperando que otra actividad finalice (e.g., acceso a cierto dispositivo externo, o acceso a otro objeto activo)
- para priorizar actividades (y potencialmente mejorar la respuesta)

- para apoyar la escalabilidad del sistema (con procesos adicionales que compartan la carga)
- para separar intereses entre las diferentes áreas del software
- para alcanzar una mayor disponibilidad del sistema (con procesos de backup)

De la lógica al desarrollo. Una clase se implementa generalmente como un modulo, por ejemplo un tipo de la parte visible de un paquete. Las clases grandes se descomponen en múltiples paquetes. Colecciones de clases íntimamente relacionadas –categorías de clases– se agrupan en subsistemas. Deben también considerarse otras restricciones para la definición de subsistemas tales como la organización del equipo de desarrollo, el tamaño esperado del código (típicamente 5K a 20K SLOC por subsistema), grado de reúso y comparación esperado, principio de distribución en capas (visibilidad), políticas de liberación, y administración de la configuración. Por lo tanto, generalmente terminamos con una vista que no tiene necesariamente una relación uno a uno con la vista lógica.

Las vistas lógica y de desarrollo son muy cercanas, aunque se refieren a distintos asuntos. Hemos encontrado que cuanto mayor es el proyecto, mayor es también la distancia entre estas dos vistas. Similarmente para las vistas de procesos y física: cuanto mayor el proyecto, mayor es la distancia entre estas vistas.

Confeccionando el Modelo

No toda arquitectura de software requiere las “4+1” vistas completas. Las vistas que no son útiles pueden omitirse de la descripción de arquitectura, tales como la vista física si hay un único procesador, y la vista de procesos si existe un solo proceso o programa. Para sistemas muy pequeños, es posible que las vistas lógica y de desarrollo sean tan similares que no requieran descripciones independientes. Los escenarios son útiles en todas las circunstancias.

1.2 Desarrollo orientado a objetos.

Enfoque de Orientación a Objetos

El enfoque de orientación a objetos es una forma de observar la realidad. El enfoque como su nombre lo indica se basa en el concepto de objeto:

Es todo aquello que tiene características que lo hacen único e indivisible dentro del entorno al que pertenece. Siempre es posible establecer propiedades o atributos de un objeto, lo mismo que su grado de respuesta a estímulos externos (comportamientos del objeto).

Normalmente interactuamos con todo tipo de objetos a nuestro alrededor y los percibimos tal cual por sus características y su respuesta ante el entorno. Un elemento notable es que las características de un objeto pueden ser por sí mismas otros objetos.

La orientación a objetos promete mejoras de amplio alcance en la forma de diseño, desarrollo y mantenimiento del software ofreciendo una solución a largo plazo a los problemas y preocupaciones que han existido desde el comienzo en el desarrollo de software: la falta de portabilidad del código y reusabilidad, código que es difícil de modificar, ciclos de desarrollo largos y técnicas de codificación no intuitivas.

Un lenguaje orientado a objetos ataca estos problemas. Tiene tres características básicas:

- basado en objetos,
- basado en clases y
- capaz de tener herencia de clases.

El elemento fundamental de la OOP es, como su nombre lo indica, el objeto. Podemos definir un objeto como un conjunto complejo de datos y programas que poseen estructura y forman parte de una organización.

Esta definición especifica varias propiedades importantes de los objetos. En primer lugar, un objeto no es un dato simple, sino que contiene en su interior cierto número de componentes bien estructurados. En segundo lugar, cada objeto no es un ente aislado, sino que forma parte de una organización jerárquica o de otro tipo.

Estructura de un Objeto

Un objeto puede considerarse como una especie de cápsula dividida en tres partes:

1. RELACIONES. Las relaciones permiten que el objeto se inserte en la organización y están formadas esencialmente por punteros a otros objetos.
2. PROPIEDADES Las propiedades distinguen un objeto determinado de los restantes que forman parte de la misma organización y tiene valores que dependen de la propiedad de que se trate. Las propiedades de un objeto pueden ser heredadas a sus descendientes en la organización.
3. METODOS. Los métodos son las operaciones que pueden realizarse sobre el objeto, que normalmente estarán incorporados en forma de programas (código) que el objeto es capaz de ejecutar y que también pone a disposición de sus descendientes a través de la herencia.

Clases e Instancias

Dentro de la definición de objetos hay algunos que serán muy parecidos ya sea que comparten la misma definición de características. Una clase es una ayuda notacional en el análisis para establecer las propiedades y funciones que serán comunes a todos los objetos que se generen a partir de ella, tal y como lo muestra la figura 1.2.

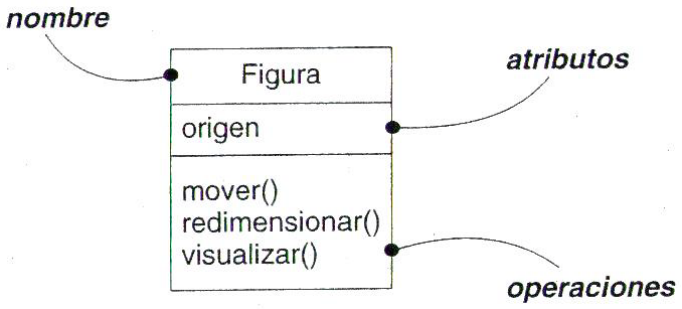


Figura 1.2 Clase en notación UML

En la notación de la metodología UML se utiliza un rectángulo con tres divisiones para definir una clase. En la primera se indica el nombre de la clase, la segunda contendrá todas las propiedades del objeto y la tercera contiene las funciones por media de las cuales el objeto responde a estímulos de su entorno.

En el enfoque orientado a objetos, todos los objetos se generan a partir de clases. Incluso cuando solo se genere uno, de forma que todo objeto está asociado a la clase a partir de la que se generó y se dice que dicho objeto es una instancia de clase que puede usar directamente las características y funciones asociadas a la clase.

La clase es un elemento conceptual que permite definir y modelar, el objeto o instancia de clase es una ejemplificación de clase. Una instancia de clase tiene valores de atributos y puede invocar sus métodos.

Encapsulamiento

Cuando interactuamos con un objeto únicamente nos interesa conocer las características y atributos que nos permiten establecer dicha interacción, es decir el objeto “publica” ciertas características y métodos, pero encapsula las demás ya que no es fundamental para el exterior conocerlas para poder interactuar con el objeto. Únicamente se requiere conocer aquellas que se exhiban como públicas.

Polimorfismo

El polimorfismo se define como la característica de que un objeto que pide la ejecución de una función a otro objeto no necesita saber la naturaleza del objeto dueño de la función. El polimorfismo más común lo tenemos en un operador como el de suma o multiplicación, ya que la operación se realiza aun cuando los operándoos sean distintos. La implementación del operador tiene el cuidado de

hacer correctamente la operación para el tipo de datos que está participando como se muestra en la figura 1.3.

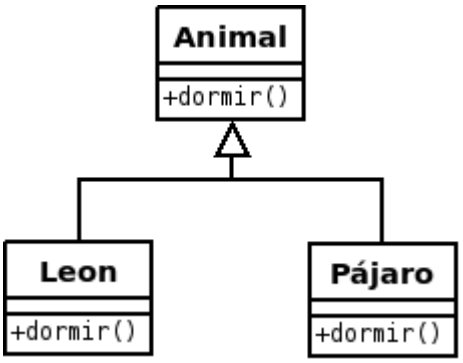


Figura 1.3. La clase animal y las subclases león y pajaro. Polimorfism

Herencia

El concepto de herencia va asociado a la Generalización –Especialización- en el que se establece que cierta clase de objetos puede tener clases de objetos más especializados las cuales por ser especializaciones, tendrán todos los elementos de la clase genérica y modificarán la funcionalidad heredada o agregarán nueva funcionalidad para especializarse.

En la notación UML la relación de herencia entre dos clases se indica por medio de una flecha hueca que apunta a la clase ascendiente, esto es porque, en una relación de herencia el padre no se construye teniendo en cuenta que se pueden derivar de él clases más especializadas. Por otro lado las clases que son descendientes es importante que sepan quién es su ascendiente pues de él están tomando propiedades y funciones. Ejemplo de herencia la figura 1.4.

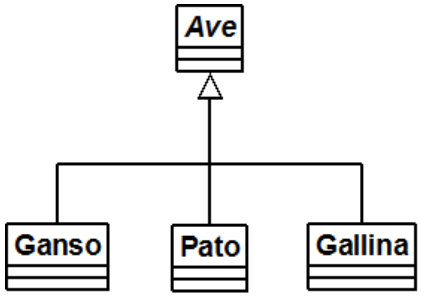


Figura1.4. Herencia de clase

Las clases que no generan directamente instancias se denominan abstractas, y aquellas que generan directamente objetos se les denominan concretas.

Asociaciones estáticas

Permiten la estructuración de los objetos incorporando reglas del negocio. En su forma más simple es un vínculo entre dos objetos. Al nivel de análisis se documentan como vínculos entre clases, sin embargo, es importante recordar que las clases son definiciones generales de un tipo de objeto, lo que implica que no son colecciones de objetos y por lo tanto las asociaciones estáticas no son asociaciones entre conjuntos de objetos como se pueden dar en el modelo Entidad Relación tradicional.

La asociación se indica con una línea uniendo a las dos clases. La línea puede tener una etiqueta que indique la forma en la que se está interpretando la asociación. Muchas reglas del negocio se refieren a los niveles máximos y mínimos de vinculación entre distintos objetos de la aplicación. Por ejemplo, un objeto Cliente solo debe estar asociado a un solo agente de Ventas, aunque un Agente Vendedor puede estar vinculado a más de un cliente. Este tipo de información es la multiplicidad o cardinalidad de la asociación y se indica colocando un rango en los extremos de la asociación.

Existen casos en los que una clase se asocia con más de una clase, en donde es conveniente no sólo mostrar la multiplicidad, sino también cómo el analista concibe el vínculo de la clase en cada asociación. En estos casos se le define un rol a la clase en a cada asociación que participa como se muestra en la figura 1.5.

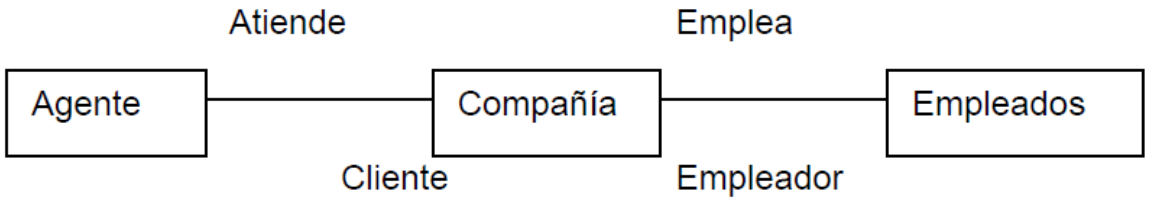


Figura 1.5: Asociaciones estáticas

Asociaciones de agregación o composición.

Un caso particular de asociación estática, es cuando se encuentra que un objeto no sólo está asociado con otro, sino que además uno es parte del otro. Este tipo de situación se le llama asociación de agregación.

La agregación podemos establecerla en dos tipos: por composición y por agregación simple como se muestra en la figura 1.6. En la primera, los objetos y la asociación se dieron simultáneamente y cuando desaparezca un objeto, el otro también desaparecerá, así como la asociación. Por lo regular se da cuando modelamos un objeto como propiedad de otro objeto.

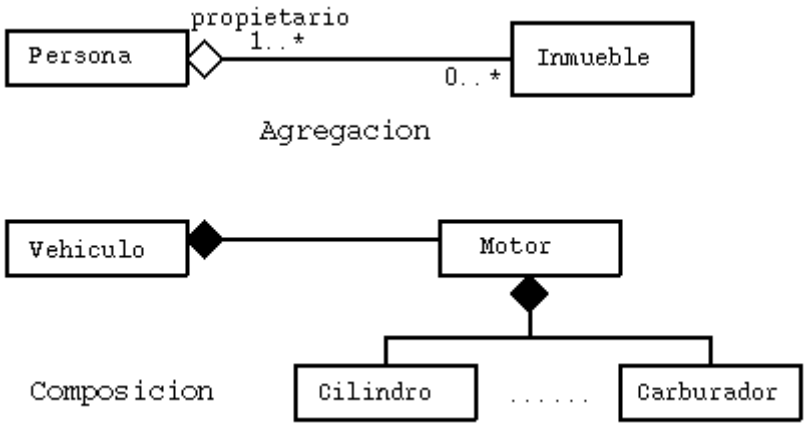


Figura 1.6 Agregación y composición.

La agregación simple es la más común y se da cuando tenemos documentos, tales como facturas, órdenes de entrada de almacén, recibos de pago, etc. Ya que el documento es el objeto central y tiene una serie de características con cierto grado de opcionalidad, que hacen no necesiten estar los dos para que la agregación se dé.

En UML la agregación se denota con un rombo en el extremo de la asociación en donde se encuentra la clase a la que se le están agregando objetos. La agregación por composición se denota con el rombo sólido, mientas que la agregación simple es con el rombo vacío.

En general es conveniente indicar las asociaciones de agregación pues sugieren que las operaciones de mantenimiento a esos objetos tienen que hacerse dentro de una misma transacción, lo cual ayuda a identificar la complejidad de los modelos de objetos que se están obteniendo. Siempre será más fácil dar mantenimiento a una familia de objetos que a dos o más familias de objetos como lo sugieren las asociaciones de agregación.

1.3 Diagramación

Modelos.

Un modelo representa a un sistema software desde una perspectiva específica. Al igual que la planta y el alzado de una figura en dibujo técnico nos muestran la misma figura vista desde distintos ángulos, cada modelo nos permite fijarnos en un aspecto distinto del sistema. Los modelos de UML que se tratan en esta parte son los siguientes:

- Diagrama de Estructura Estática.
- Diagrama de Casos de Uso.
- Diagrama de Secuencia.
- Diagrama de Colaboración.

- Diagrama de Estados.

Diagramas de clases

Los diagramas de clases proporcionan una perspectiva estática del sistema (representan su diseño estructural), son los bloques de construcción más importantes de cualquier sistema orientado a objetos y es una descripción de un conjunto de objetos que comparten los mismos atributos, operaciones, relaciones y semántica.

Como muestra la Figura 1.7 Esta notación (UML) permite visualizar una abstracción independientemente de cualquier lenguaje de programación específico y de forma que permite resaltar las partes más importantes de una abstracción: su nombre, sus atributos y sus operaciones.

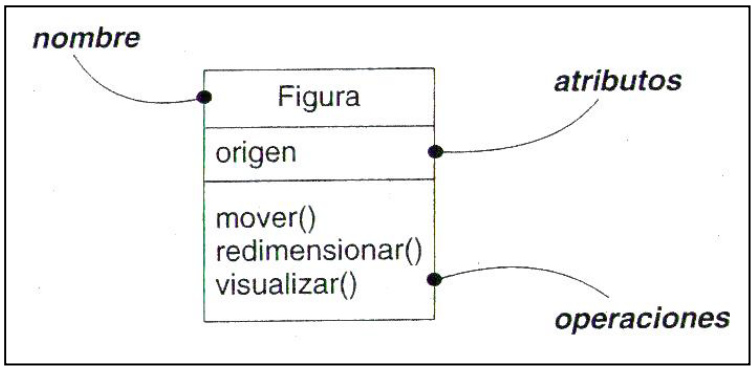


Figura 1.7: Clase

Cada clase ha de tener un nombre que la distinga de otras clases. El Nombre es una cadena de texto. Ese nombre sólo se denomina nombre simple. Una clase puede dibujarse mostrando sólo su nombre.

Un atributo es una propiedad de una clase identificada con un nombre, que describe un rango de valores que pueden tomar las instancias de la propiedad.

Una operación es la implementación de un servicio que puede ser requerido a cualquier objeto de la clase para que muestre un comportamiento, En otras palabras, una operación es una abstracción de algo que se puede hacer a un objeto y que es compartido por todos los objetos de la clase, Una clase puede tener cualquier número de operaciones o ninguna, en la figura 1.8 muestra el diagrama completo del dominio conceptual de una aplicación.

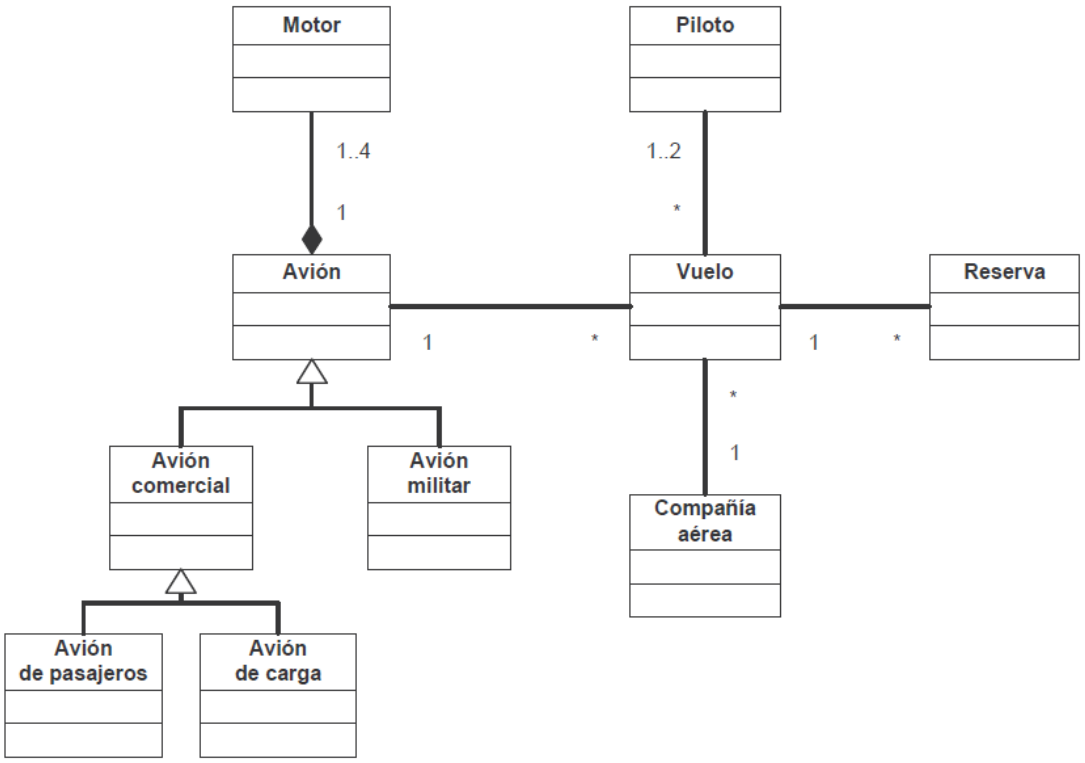


Figura. 1.8. Diagrama de clases

Diagrama de Casos de Uso.

Un Diagrama de Casos de Uso muestra la relación entre los actores y los casos de uso del sistema. Representa la funcionalidad que ofrece el sistema en lo que se refiere a su interacción externa. En el diagrama de casos de uso se representa también el sistema como una caja rectangular con el nombre en su interior como se muestra en la figura 1.9. Los casos de uso están en el interior de la caja del sistema, y los actores fuera, y cada actor está unido a los casos de uso en los que participa mediante una línea.

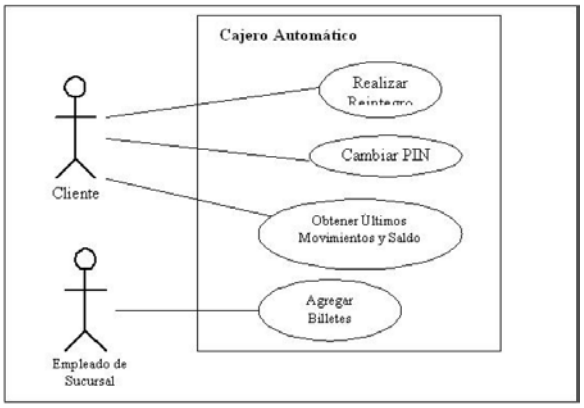


Figura 1.9: Diagrama de casos de uso

Diagramas de Interacción.

En los diagramas de interacción se muestra un patrón de interacción entre objetos. Hay dos tipos de diagrama de interacción, ambos basados en la misma información, pero cada uno enfatizando un aspecto particular: Diagramas de Secuencia y Diagramas de Colaboración.

Un diagrama de Secuencia muestra una interacción ordenada según la secuencia temporal de eventos. En particular, muestra los objetos participantes en la interacción y los mensajes que intercambian ordenados según su secuencia en el tiempo. Figura 1.10.

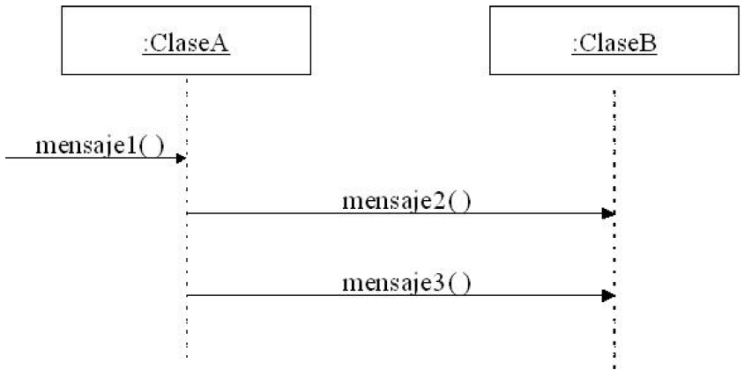


Figura 1.10: Diagrama de secuencia

Un Diagrama de Colaboración muestra una interacción organizada basándose en los objetos que toman parte en la interacción y los enlaces entre los mismos (en cuanto a la interacción se refiere véase figura 1.11). A diferencia de los Diagramas de Secuencia, los Diagramas de Colaboración muestran las relaciones entre los roles de los objetos. La secuencia de los mensajes y los flujos de ejecución concurrentes deben determinarse explícitamente mediante números de secuencia.

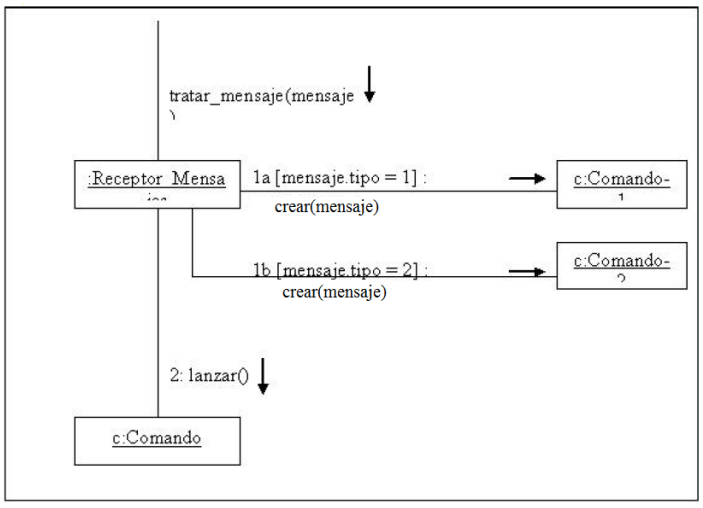


Figura 1.11: Diagrama de colaboración

En cuanto a la representación, un Diagrama de Colaboración muestra a una serie de objetos con los enlaces entre los mismos, y con los mensajes que se intercambian dichos objetos. Los mensajes son flechas que van junto al enlace por el que “circulan”, y con el nombre del mensaje y los parámetros (si los tiene) entre paréntesis. Cada mensaje lleva un número de secuencia que denota cuál es el mensaje que le precede, excepto el mensaje que inicia el diagrama, que no lleva número de secuencia. Se pueden indicar alternativas con condiciones entre corchetes.

Diagramas de Estado.

Un Diagrama de Estados muestra la secuencia de estados por los que pasa bien un caso de uso, bien un objeto a lo largo de su vida, o bien todo el sistema. En él se indican qué eventos hacen que se pase de un estado a otro y cuáles son las respuestas y acciones que genera.

En cuanto a la representación, un diagrama de estados es un grafo cuyos nodos son estados y cuyos arcos dirigidos son transiciones etiquetadas con los nombres de los eventos.

Un estado se representa como una caja redondeada con el nombre del estado en su interior. Una transición se representa como una flecha desde el estado origen al estado destino.

La caja de un estado puede tener 1 o 2 compartimentos. En el primer compartimento aparece el nombre del estado véase figura 1.12. El segundo compartimento es opcional, y en él pueden aparecer acciones de entrada, de salida y acciones internas.

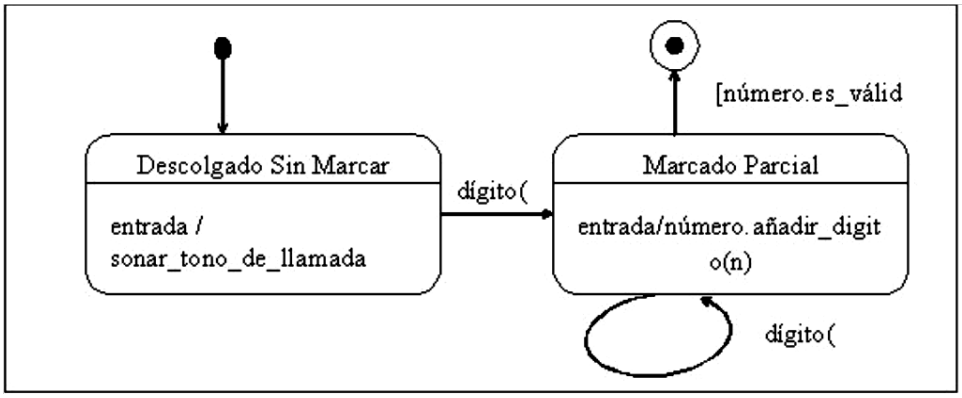


Figura 1.12: Diagrama de Estados

Modelado Físico De Un Sistema OO.

<u>Componentes.</u>

Los componentes pertenecen al mundo físico, es decir, representan un bloque de construcción al modelar aspectos físicos de un sistema. Una característica básica de un componente es que debe definir una abstracción precisa con una interfaz bien definida, y permitiendo reemplazar fácilmente los componentes más viejos con otros más nuevos y compatibles."

En UML todos los elementos físicos se modelan como componentes. UML proporciona una representación gráfica para estos como se puede ver en la Figura 1.13, en la que XXXX.dll, es el nombre del componente.

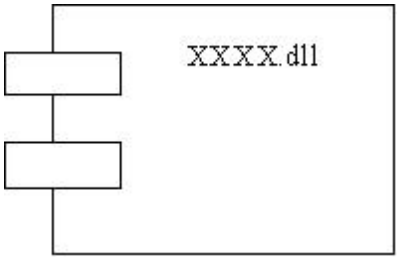


Figura 1.13. Componente físico

Cada componente debe tener un nombre que lo distinga de los demás. Al igual que las clases los componentes pueden enriquecerse con compartimentos adicionales que muestran sus detalles, como se puede ver en la Figura 1.14.

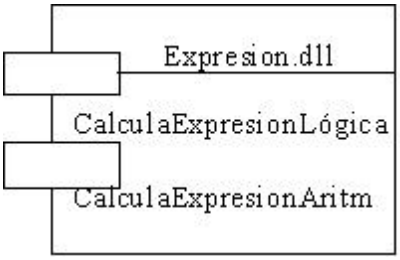


Figura 1.14. Componente físico especifico.

Agrupación de Elementos Mediante Paquetes

Un paquete es un mecanismo de propósito general para organizar elementos en grupos. Cualquier grupo de elementos, sean estructurales o de comportamiento, puede incluirse en un paquete. Incluso pueden agruparse paquetes dentro de otro paquete.

Un paquete se representa como un rectángulo grande con un pequeño rectángulo sobre la esquina superior izquierda a modo de lengüeta. Si no se muestra el contenido del paquete entonces el nombre del paquete se coloca dentro del rectángulo grande. Si, por el contrario, se quiere mostrar el contenido del paquete,

entonces el contenido va dentro del rectángulo grande y el nombre del paquete va en la lengüeta.

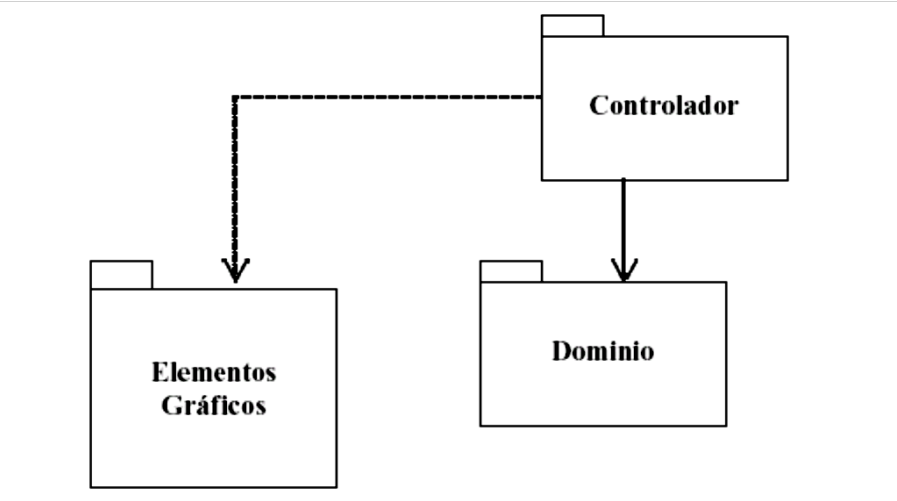


Figura 1.15. Agrupación por Paquetes.

Se pueden indicar relaciones de dependencia entre paquetes mediante una flecha con la línea a trazos. como se muestra en la figura 15.

2. Diseño orientado a objetos

2.1 Diseño del sistema en base a procesos

La primera actividad en el proceso de desarrollo de sistemas involucra al equipo de trabajo en el conocimiento del problema para establecer los requerimientos en términos confiables y especificaciones formales y concretas.

El proceso de análisis de requerimientos pretende acelerar las curvas de aprendizaje mediante subprocesos que permitan extraer el conocimiento clave de los usuarios hacia estructuras incrementales de conocimiento, de tal forma que los miembros del equipo de desarrollo, puedan llegar más rápidamente a la parte acelerada de la curva de aprendizaje.

Sin embargo, ésta adquisición de conocimientos no será sólo capacitación, sino también, debe aprovecharse para establecer los requerimientos básicos del sistema.

El Modelo de Requerimientos se constituye a su vez de tres modelos:

- Modelo de casos
- Modelo de Interfaz
- Modelo de Dominio del Problema

Cada uno contempla parte del conocimiento del usuario orientado a la especificación de las funciones básicas del sistema.

El Modelo de Casos extrae el conocimiento funcional fundamental del problema de una forma estructurada y progresiva, siendo la base para establecer la estructura del sistema. Este modelo orienta todos las demás procesos del método.

El modelo de Interfaz establece el vínculo visual entre el desarrollador y el usuario para concretar aspectos de la interacción que el sistema pudiese tener con su entorno externo, permitiendo la retroalimentación temprana de aspectos fundamentales para el conocimiento de la aplicación.

En el Modelo del Dominio del Problema se establecerán los principales objetos que constituirán al sistema y las relaciones que tienen entre sí.

2.1.1 Actividades y casos de uso.

Modelado de casos de uso

Un caso real de uso describe el diseño concreto del caso de uso a partir de una tecnología particular de entrada y salida, así como de su implementación global. Por ejemplo, si interviene una interfaz gráfica para el usuario, el caso de uso real incluirá diagramas de las ventanas en cuestión y una explicación de la interacción de bajo nivel con los artefactos de la interfaz.

Los diagramas de casos de uso pueden llegar a ser bastante grandes esto no es malo con los modelos grandes, siempre y cuando se agregue valor al esfuerzo global. El hecho es que se puede llegar a crear un modelo de gran tamaño que describe los requisitos para su sistema, y puede incluir un modelo de casos de uso general que se han creado durante meses o años de trabajo evolutivo. No deben ser creados todos por adelantado antes de empezar a programar.

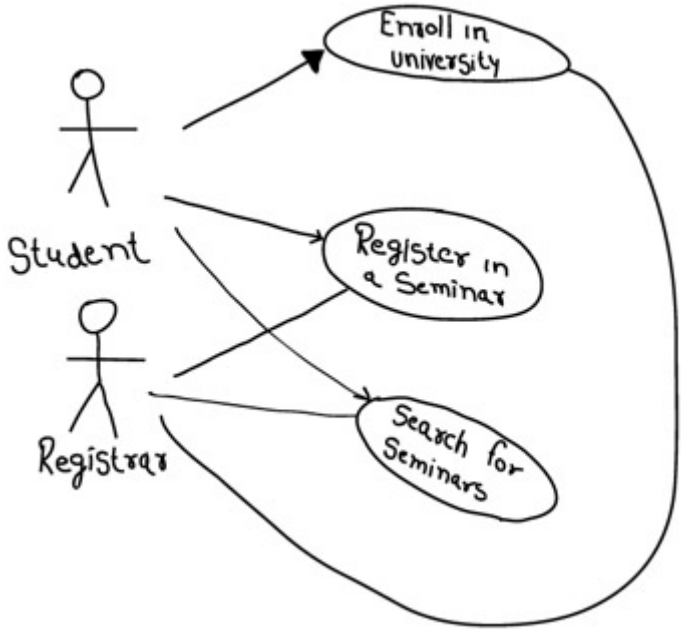


Figura 2.1: Boceto inicial de un diagrama de casos de uso.

Ahora veamos lo que el simple diagrama de la figura. 2.1 podría evolucionar. El diagrama de casos de uso en la figura 2.2. Abajo proporciona un ejemplo de un diagrama de casos de uso UML.:

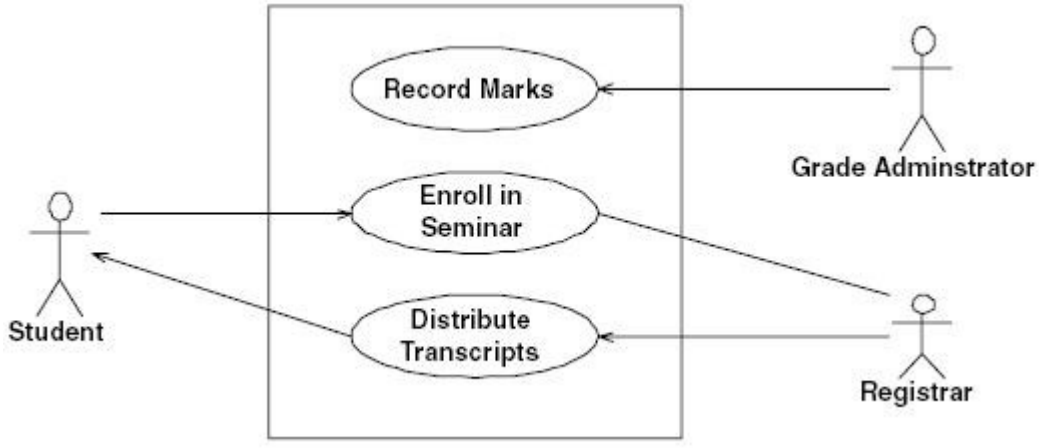


Figura 2.2: Un simple diagrama de casos de uso para una universidad

En el ejemplo representado en la figura 2.2. Los estudiantes se matriculan en cursos con la ayuda del encargado de admisión. Los profesores ingresan las notas obtenidas por los estudiantes de las actividades asignadas y el encargado administrador autoriza la publicación de calificaciones a los estudiantes.

La asociación entre el estudiante e inscrito en seminario señala que el caso de uso es invocado, inicialmente por un estudiante y no por un encargado de admisión (el encargado de admisión es también un actor y está involucrado en este caso de uso). Entendiendo que las asociaciones no representan flujo de información es importante, simplemente indican que un actor es de alguna manera involucrado en un caso de uso. La información fluye de ida y vuelta entre el actor y el caso de uso, por ejemplo, los estudiantes deben indicar a qué seminarios quieren inscribirse en el sistema y tendría que indicar a los estudiantes si han sido inscritos.

Figura 2.3: Diagrama de casos de uso más complejo de la misma universidad

Los Diagramas de casos de uso dan una visión general muy buena de los requisitos. A menudo se dibuje un diagrama de casos de uso mientras se están identificando los actores del caso de uso y las asociaciones entre ellos.

(Schneider y Winters, 2001) sugieren que los diagramas de casos de uso deben ser desarrollados desde el punto de vista del usuario y no del desarrollador. Su

objetivo es modelar los requisitos de comportamiento para el sistema y cómo los usuarios trabajan con el sistema para satisfacer sus necesidades, no lo que los desarrolladores creen que deben construir.

Identificación de los actores

Un actor representa a cualquier cosa o alguien que interactúa con el sistema. Esto puede incluir a las personas (no sólo el usuario final), los sistemas externos, y otras organizaciones. Los actores son siempre externos al sistema que está siendo modelado, ellos nunca son parte del sistema. Para ayudar a encontrar a los actores en el sistema, debe hacerse las siguientes preguntas (Schneider y Winters 2001; Leffingwell y Widrig 2000):

¿Quién es el principal cliente de su sistema?

¿Quién obtiene la información de este sistema?

¿Quién proporciona información al sistema?

¿Quién instala el sistema?

¿Quién opera el sistema?

¿Quién apaga el sistema?

¿Qué otros sistemas interactúan con este sistema?

¿Hay algo que sucederá de forma automática a una hora predeterminada?

¿Quién suministro, utilización, o eliminar la información del sistema?

¿De dónde viene el sistema de obtener información?

Un caso de uso es una secuencia de acciones que proporcionen un valor medible para un actor. Otra forma de verlo es un caso de uso es, describe una manera en la que un actor del mundo real interactúa con el sistema.

Identificación de casos de uso

Una forma en preguntando a sus interesados las siguientes preguntas desde el punto de vista de los actores:

¿Cuáles son los usuarios de este rol tratando de realizar?

¿Qué deben ser capaces los usuarios de hacer?

¿Cuáles son las principales tareas de los usuarios en este papel?

¿Qué información necesitan examinar, crear o cambiar los usuarios de este perfil?

¿Qué información necesitan los usuarios de este perfil del sistema?

¿Qué hacen los usuarios en este perfil que necesitan informar al sistema?

Los sistemas y sus fronteras

Un caso de uso describe la interacción con un "sistema". Las fronteras ordinarias del sistema son:

- la frontera hardware/software de un dispositivo o sistema de cómputo
- el departamento de una organización
- la organización entera

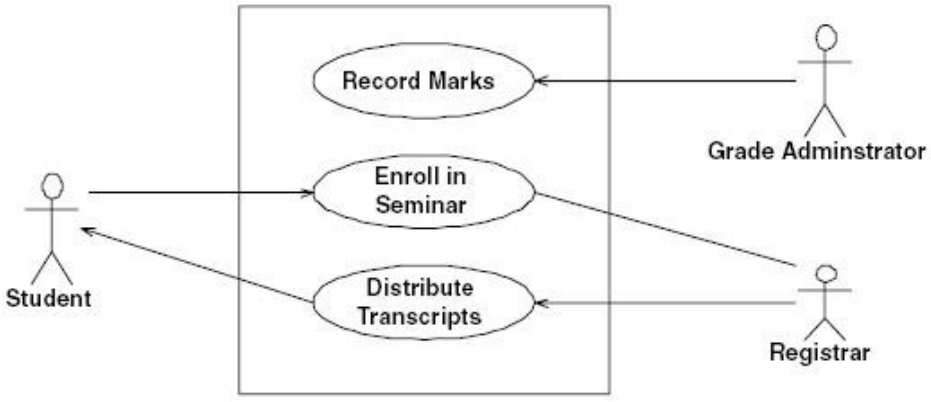


Figura 2.4 fronteras de un sistema

Es importante definir la frontera del sistema como se muestra en la figura 2.4, para identificar lo que es interno o externo, así como las responsabilidades del sistema. El ambiente externo está representado únicamente por actores.

Casos de uso primario, secundario y opcional.

Los casos deberían clasificarse en primarios, secundarios u opcionales. Más adelante, a partir de estas designaciones, clasificaremos nuestro conjunto de casos de uso para establecer prioridades en su desarrollo.

Los casos primarios de uso representan los procesos comunes más importantes, como Comprar productos.

Los casos secundarios de uso representan procesos menores o raros; por ejemplo,

Los casos opcionales de uso representan procesos que pueden no abordarse.

2.1.2 Interfaz de usuario

Un caso real de uso describe el diseño concreto del caso de uso a partir de una tecnología particular de entrada y salida, así como de su implementación global. Por ejemplo, si interviene una interfaz gráfica para el usuario, el caso de uso real incluirá diagramas de las ventanas en cuestión y una explicación de la interacción de bajo nivel con los artefactos de la interfaz.

Una alternativa podría consistir en que el diseñador realizara storyboards o secuencias de las pantallas de la interfaz general para el usuario y después incorporar los detalles durante la implementación.

Los storyboards son de gran utilidad, si antes de la implementación los diseñadores o el cliente necesitan descripciones rigurosamente detalladas.

La interfaz de usuario UI[2] es la parte del software con el que un usuario interactúa directamente. Un prototipo de interfaz de usuario es un modelo de baja fidelidad de la interfaz de usuario para el sistema. Representa las ideas generales de la IU, pero no los detalles exactos. Esencialmente los prototipos de IU representan los requisitos de una manera independiente de la tecnología, al igual que los modelos de casos de uso esenciales para hacer sus necesidades conductuales. Un prototipo de IU es realmente el estado inicial, el inicio de la interfaz para el sistema. Estos modelos de requisitos de interfaz de usuario, evolucionan a través del análisis y diseño para dar lugar a la interfaz de usuario final para su sistema.

Cuando un equipo está creando un elemento esencial del prototipo de interfaz de usuario, este itera entre las siguientes tareas:

Explorar el uso del sistema. Su equipo explorará el uso del sistema a través de diversos medios.

Modelo de los elementos principales de la IU. Elementos de la interfaz, como las pantallas de informes potenciales, se puede modelar usando papel de rota folio.

Modelo de elementos menores de la IU. Tales como campos de entrada, las listas, y los contenedores (elementos de la IU que se agregan otros elementos de menor importancia.

[2] UI del acrónimo ingles User Interface

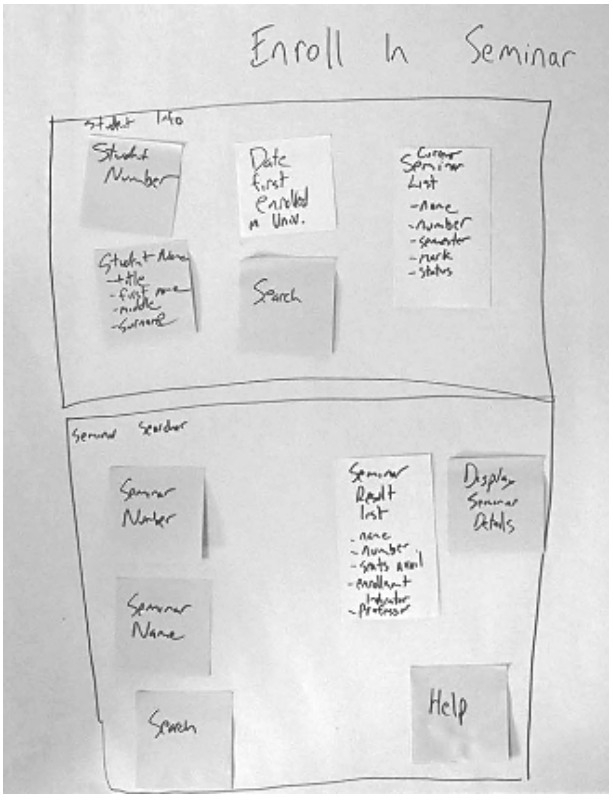


Figura 2.5: Un prototipo de interfaz de usuario.

La creación de un prototipo de interfaz de usuario es una técnica de análisis iterativo en el que los usuarios participan activamente en el bosquejo en marcha de la interfaz de usuario para un sistema como se muestra en la figura 2.5. Los prototipos de interfaz de usuario tienen varios propósitos:

- Como el análisis de un artefacto que le permite explorar el espacio del problema con las partes interesadas;
- Como un artefacto de diseño que le permite explorar el espacio de la solución de su sistema;
- Como un vehículo para que usted se comunique el posible diseño de la interfaz de usuario (s) de su sistema, y
- Como un fundamento potencial desde donde continuar desarrollando el sistema.

Mientras se están determinando las necesidades de los stakeholders se puede decidir transformar sus prototipos de interfaz de usuario, si ha creado bocetos. Figura 2.6 que representa un esquema de dos pantallas potenciales o páginas HTML basado en el prototipo de interfaz de usuario figura. 2.5. Transformar en realidad no es la palabra aquí, ya que estoy usando una tecnología de de modelización completamente diferente (una pizarra en lugar de papel) por lo que en efecto se está reemplazando lo esencial del prototipo de interfaz de usuario con los bocetos.

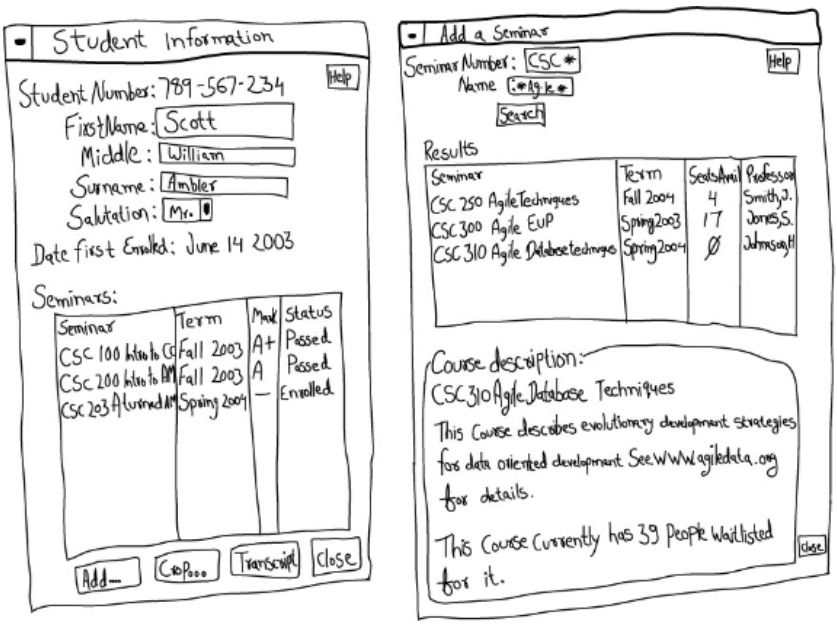


Figura 2.6: esquema de la pantalla para inscribirse en un seminario.

Una vez que entienda las necesidades de la interfaz de usuario de sus stakeholders, el siguiente paso es construir en realidad un prototipo. Usando una herramienta de creación de prototipos o lenguaje de alto nivel como Java .Net u otro, se desarrolla la pantalla, páginas y los informes que necesitan los usuarios. Con la plataforma de interfaz de usuario seleccionada, puede iniciar la conversión de los aspectos individuales de su interfaz de usuario. Es posible que desee crear bocetos, como se ve en la figura. 2.6 o ir directamente a una aplicación concreta, como la página HTML que se muestra en la figura 2.7. Los dibujos son más inclusivos, y sus partes interesadas pueden participar activamente en su creación, a pesar de que la actual página HTML es mucho más cercana al código de trabajo.

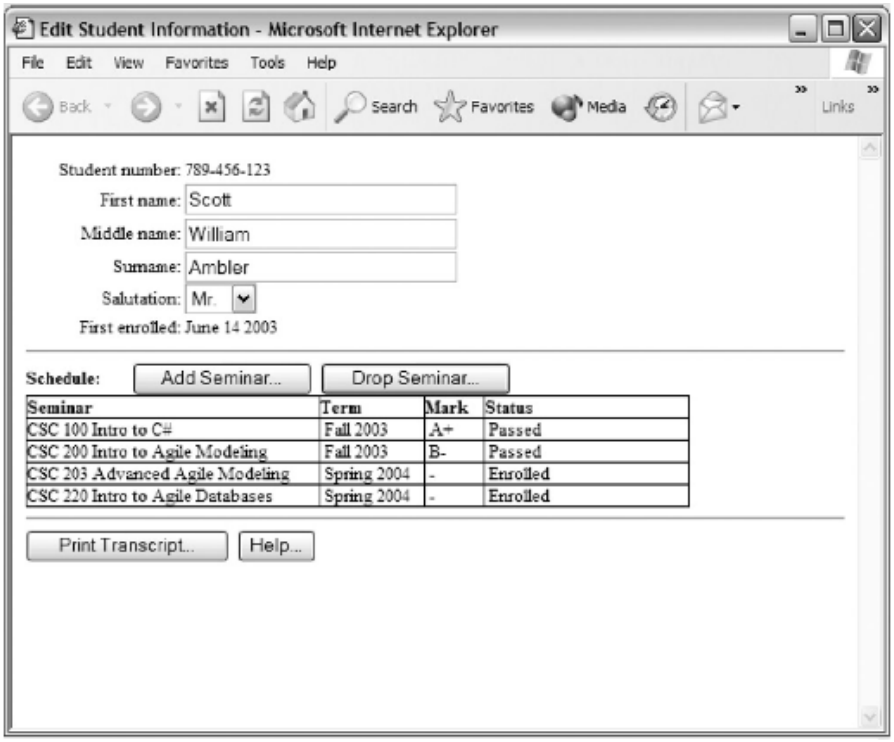


Figura 2.7. Una página HTML para inscribirse en un seminario.

Es fundamental comprender que no es necesario crear un prototipo para todo el sistema. Es muy común que al prototipo de una pequeña porción de la interfaz de usuario, tal vez una sola pantalla o la página HTML, antes de pasar a su aplicación.

2.2 Diseño de la lógica.

2.2.1 Clases y Objetos.

En un caso de uso conducido por Modelado de objetos con UML se describen una técnica llamada análisis de robustez, algo que el Rational Unified Process (RUP) se refiere a su uso como análisis de casos (IBM 2003). La idea básica es que usted puede analizar los pasos de un caso de uso para validar la lógica de negocio dentro de él y garantizar que la terminología es consistente con otros casos de uso que ha analizado anteriormente. En otras palabras, usted lo puede utilizar para asegurarse de que sus casos de uso son lo suficientemente robustos para representar los requisitos de uso para el sistema que estamos construyendo. Otro uso es el de identificar objetos potenciales o responsabilidades de objeto para apoyar la lógica en el caso de uso, actuando efectivamente como un puente hacia otros diagramas tales como diagramas de secuencia UML y los diagramas de clases UML.

Un modelo conceptual explica (a sus creadores) los conceptos significativos en un dominio del problema; es el artefacto más importante a crear durante el análisis orientado a objetos.

Una cualidad esencial que debe ofrecer un modelo conceptual es que representa cosas del mundo real, no componentes del software.

Una de las primeras actividades centrales de un ciclo de desarrollo consiste en crear un modelo conceptual para los casos de uso del ciclo actual. Esto no puede hacerse si no se cuentan con los casos y con otros documentos que permitan identificar los conceptos (objetos). La creación no siempre es lineal; por ejemplo, el modelo conceptual puede formularse en paralelo con el desarrollo de los casos de uso.

El paso esencial de un análisis o investigación orientado a objetos es descomponer el problema en conceptos u objetos individuales: las cosas que sabemos. Un modelo conceptual es una representación de conceptos en un dominio del problema (Fowler, 1996). En UML, se ilustra con un grupo de diagramas de estructura estática donde no se define ninguna operación. La designación de modelo conceptual ofrece la ventaja de subrayar fuertemente una concentración en los conceptos del dominio, no en las entidades del software.

Puede mostrarnos:

- conceptos
- asociaciones entre conceptos
- atributos de conceptos

En la figura 2.8, vemos un modelo conceptual parcial del dominio de una tienda y las ventas. Explica gráficamente que el concepto de Pago y Venta son importantes en este dominio del problema, ya que Pago se relaciona con Venta en una forma que conviene señalar y qué Venta tiene fecha y hora.

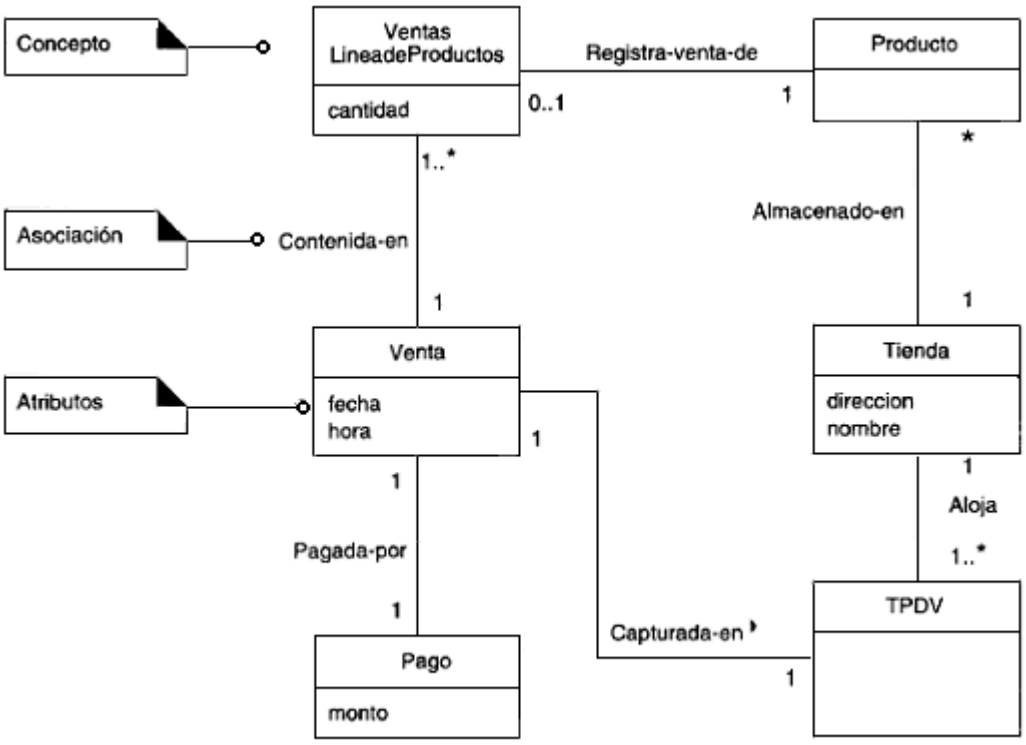


Figura 2.9 Modelo conceptual parcial.

El modelo del dominio es un diccionario visual de abstracciones; visualiza y relaciona algunas palabras o clases conceptuales del dominio. También describe una abstracción de las clases conceptuales, porque hay muchas cosas que uno podría comunicar. El modelo muestra una vista parcial, o abstracción, e ignora detalles sin interés (para el modelador).

La información que se presenta (utilizando la notación UML) podría, de manera alternativa haberse expresado en prosa, mediante sentencias en el glosario o en algún otro sitio. Pero es fácil entender los distintos elementos y sus relaciones mediante este lenguaje visual, puesto que un porcentaje significativo del cerebro toma parte en el procesamiento visual que es una cualidad de los humanos.

Por tanto, el modelo del dominio podría considerarse como un diccionario visual de las abstracciones relevantes, vocabulario del dominio e información del dominio

Un modelo del dominio, como se muestra en la Figura 2.10, es una representación de las cosas del mundo real del dominio de interés, no de componentes software, como una clase Java o C++ (ver Figura 2.11), u objetos software con responsabilidades. Por tanto, los siguientes elementos no son adecuados en un modelo del dominio:

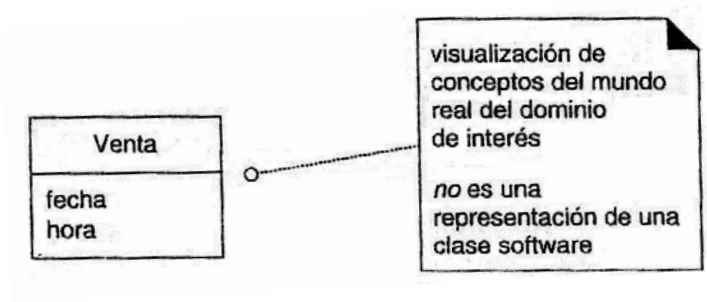


Figura 2.10 Modelo del dominio muestra clases conceptuales del mundo real.

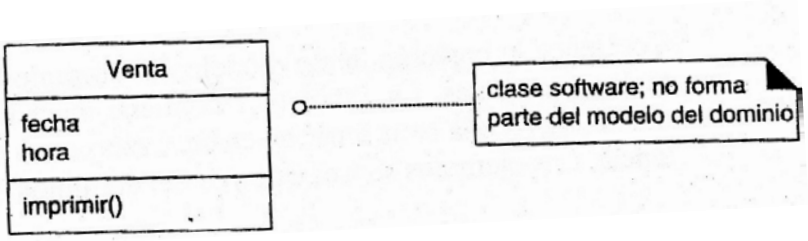


Figura 2.11 Un modelo del dominio no muestra artefactos software o clases

El modelo del dominio muestra las clases conceptuales o vocabulario del dominio. Informalmente, una clase conceptual es una idea, cosa u objeto. Más formalmente, una clase conceptual podría considerarse en términos de su símbolo, intensión, y extensión.

- Símbolo: palabras o imágenes que representan una clase conceptual.
- Intensión: la definición de una clase conceptual.
- Extensión: el conjunto de ejemplos a los que se aplica la clase conceptual.

Categorías de clases conceptuales

Se comienza la creación de un modelo del dominio haciendo una lista de clases conceptuales candidatas. La Tabla 2.1, contiene muchas categorías habituales que, normalmente, merece la pena tener en cuenta, aunque 'no en ningún orden particular de importancia.

Categoría de clase conceptual	Ejemplos
objetos tangibles o físicos	Registro Avión
especificaciones, diseños o descripciones de la cosas	EspecificacionDelProducto DescripcionDelVuelo
Lugares	Tienda
Transacciones	Venta, Pago Reserva
líneas de la transacción	LineaDeVenta
roles de la gente	Cajero,

	Piloto
contenedores de otras cosas	TIenda, Lata Avion
cosas en un contenedor	Articulo Pasajero
otros sistemas informáticos o electromecánicos externos al sistema	SistemaAutorizacionPagoCredito ControlDeTraficoAereo
conceptos abstractos	Ansia, Acrofobia
Organizaciones	DepartamentoDeVentas CompañiaAerea
Hechos	Venta, Pago, Reunion Vuelo, Colision, Aterrizaje

Tabla 2.1 categorías de clases conceptuales

Tener una declaración vaga no es muy poco común al comenzar proyectos reales. Sin embargo, debemos ser capaces construir un diagrama de clases fácilmente usando nuestros conocimientos adecuados acerca de los conceptos habituales.

Por lo tanto, iniciamos por la identificación de dos clases:

Marcador y la pluma, que tienen un número de características comunes (color, marca, etc.) Sin embargo también están las diferencias (véase, por ejemplo, que los marcadores tienen una tapa, mientras que las plumas tienen una punta retráctil). Alguien que está modelando ve de inmediato la posibilidad de una relación de generalización/especialización. Por lo tanto, introduce una clase abstracta para excluir características comunes. El modelo puede verse como en la figura 2.12:

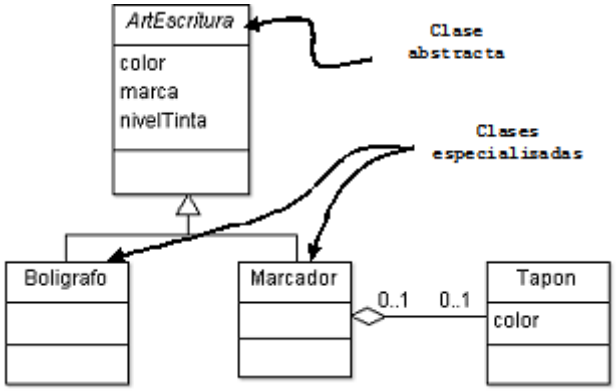


Figura 2.12. Jerarquía de herencia y agregacion

Tome nota de las multiplicidades entre marcador y tapón: un marcador puede perder su tapón, y un tapón el cuerpo de su marcador origen. Esta noción de

cuerpo es interesante y compartida para los bolígrafos y marcadores. En aras de mantener la coherencia con el tapón, hay que agregarlo.

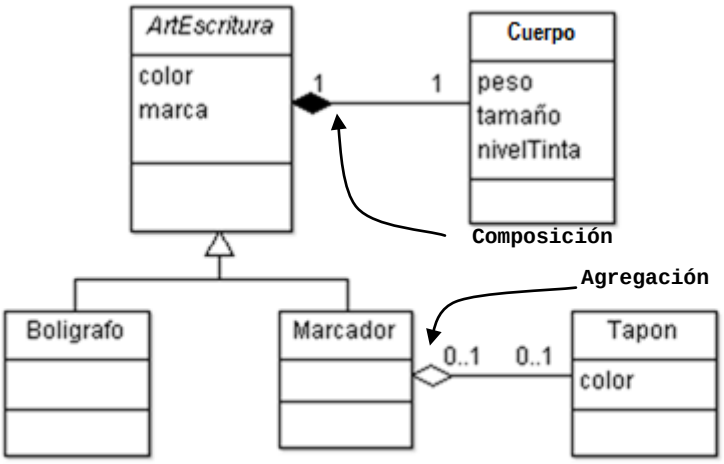


Figura 2.13.Segunda versión del diagrama de clases

La relación entre Cuerpo y ArtEscritura es de composición. Sin embargo, no es necesariamente la coherencia de los ciclos de vida entre marcador y tapon. También tenga en cuenta que el atributo nivelTinta fue trasladado al cuerpo de la clase. Este desplazamiento del atributo de una clase a otra es común especialmente en el caso de relaciones de composición o agregación, de modo que las clases son más homogéneas y coherentes.

A fin de complementar nuestro modelo, se introduce el concepto de marcador con una corrección de incorporada y sobre todo una clase para modelar el usuario y / o propietario del artículo de escritura.

2.2.2 Interacción.

Los diagramas de interacción son modelos que describen la manera en que colaboran grupos de objetos para cierto comportamiento.

Habitualmente, un diagrama de interacción capta el comportamiento de un solo caso de uso. El diagrama muestra cierto número de ejemplos de objetos y los mensajes que se pasan entre estos objetos dentro del caso de uso.

Ejemplo: enfoque mediante un caso de uso simple que exhibe el comportamiento siguiente:

- La ventana Entrada de pedido envía un mensaje "prepara" a Pedido.
- El Pedido envía entonces un mensaje "prepara" a cada Línea de pedido dentro del Pedido.

- Cada Línea de pedido revisa el Artículo de inventario correspondiente.
    - Si esta revisión devuelve "verdadero", la Línea de pedido descuenta la cantidad apropiada de Artículo de inventario del almacén.
    - Si no sucede así, quiere decir que la cantidad del Artículo de inventario ha caído más abajo del nivel de reorden y entonces dicho Artículo de inventario solicita una nueva entrega.

Hay dos tipos de diagramas de interacción: diagramas de secuencia y diagramas de colaboración.

Diagramas de secuencia

En un diagrama de secuencia, un objeto se muestra como caja en la parte superior de una línea vertical punteada (véase la figura 2.14).

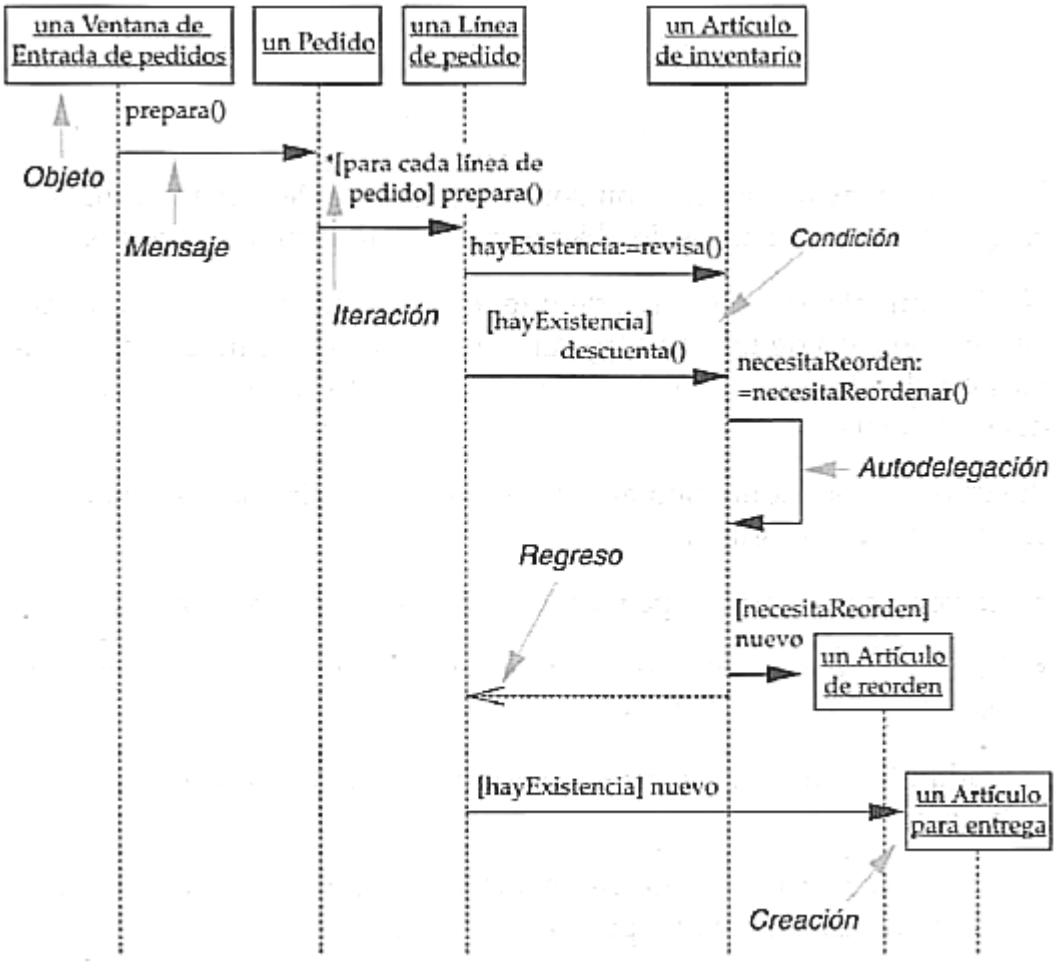


Figura 2.14: Diagrama de secuencia

Esta línea vertical punteada se llama línea de vida del objeto. La línea de vida representa la vida del objeto durante la interacción.

Cada mensaje se representa mediante una flecha entre las líneas de vida de dos objetos. El orden en el que se dan estos mensajes transcurre de arriba hacia abajo. Cada mensaje es etiquetado por el nombre del mensaje; pueden incluirse los argumentos y alguna información de control, y se puede mostrar la autodelegación (acción recursiva), que es un mensaje que un objeto se envía a sí mismo, regresando la flecha de mensaje de vuelta a la misma línea de vida.

Dos partes de la información de control son valiosas. Primero, hay una condición, que indica cuándo se envía un mensaje (por ejemplo, [necesitaReorden). El mensaje se envía sólo si la condición es verdadera.

El segundo marcador de control útil es el marcador de iteración, que muestra que un mensaje se envía muchas veces a varios objetos receptores, corno sucedería cuando se itera sobre una colección. La base de la iteración se puede mostrar entre corchetes (como en *[para cada línea de pedido]).

Corno se puede apreciar, la figura 2.14 es muy simple y tiene un atractivo visual inmediato; en ello radica su gran fuerza.

Una de las cuestiones más difíciles de comprender en un programa orientado a objetos es el flujo de control general. Un buen diseño tiene muchísimos pequeños métodos en diferentes clases, y a veces resulta muy complicado determinar la secuencia global de comportamiento.

Podrá acabar leyendo el código, tratando de encontrar dónde está el programa. Esto ocurre así, en especial, para todos aquellos que comienzan a trabajar con objetos. Los diagramas de secuencia le ayudan a ver la secuencia.

Este diagrama incluye un regreso, el cual indica el regreso de un mensaje, no un nuevo mensaje. Los regresos difieren de los mensajes normales en que la línea es punteada.

En la figura 2.15 vemos algunos objetos que revisan una transacción bancaria. Cuando se crea una Transacción, ésta crea un Coordinador de transacción que coordina el trámite de la Transacción. Este coordinador crea una cantidad (en el presente caso, dos) de objetos Revisor de transacción, cada uno de los cuales es responsable de una revisión en particular.

Este proceso permitirá añadir con facilidad otros procesos de revisión, porque cada registro es llamado asincrónicamente y opera en paralelo.

Cuando termina un Revisor de transacción, se lo notifica al Coordinador de transacción. El coordinador comprueba si todos los revisores respondieron; de no ser así, no hace nada. Si todos han respondido indicando terminaciones exitosas,

como en el presente caso, entonces el coordinador notifica a la Transacción que todo está bien.

La figura 2.15, introduce varios elementos nuevos en los diagramas de secuencia. En primer lugar, se ven las activaciones, que aparecen explícitamente cuando está activo un método, ya sea porque está efectuando operaciones o porque se encuentra esperando la devolución de una subrutina

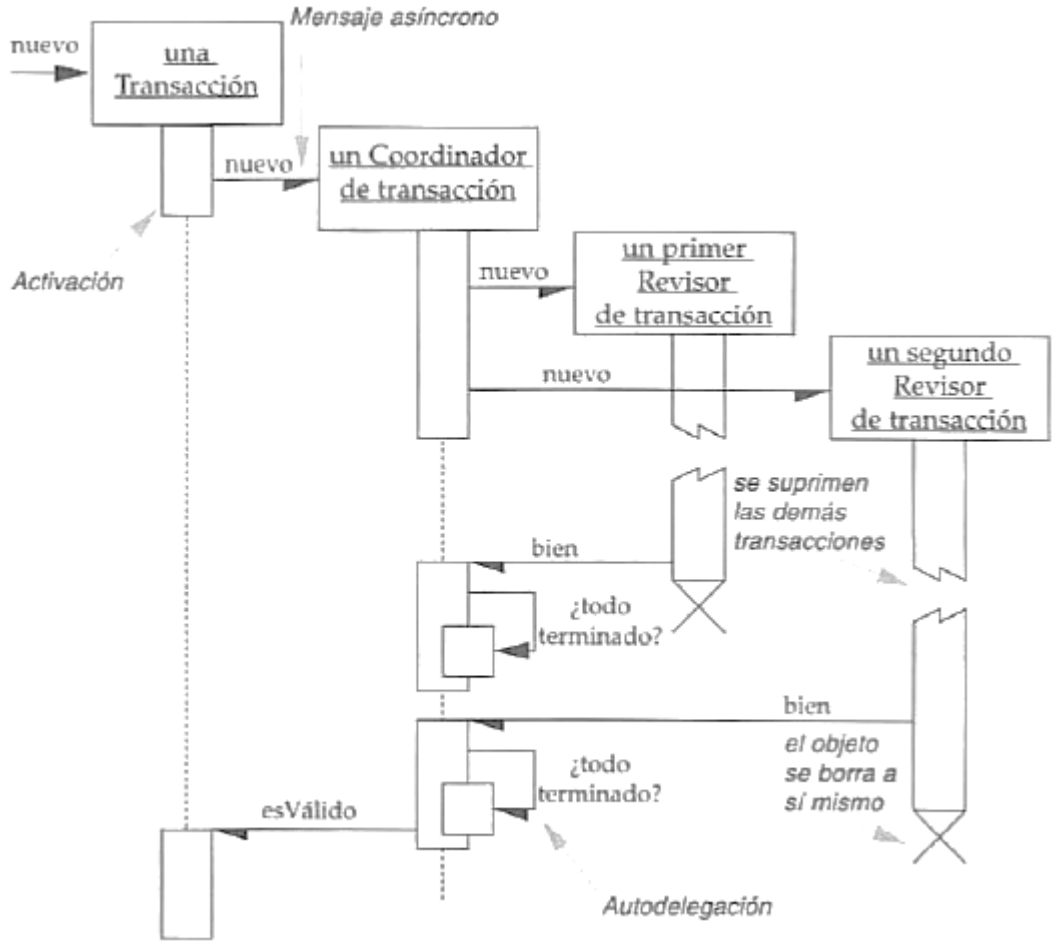


Figura 2.15 diagrama de secuencia ampliado.

Las medias cabezas de flecha indican mensajes asíncronos. Un mensaje asíncrono no bloquea al invocador, por lo cual puede continuar con su proceso. Un mensaje asíncrono puede hacer alguna de estas tres cosas:

1. Crear un nuevo proceso, en cuyo caso se vincula con el principio de una activación.
2. Crear un nuevo objeto.
3. Comunicarse con un proceso que ya está operando.

El borrado (deletion) de un objeto se muestra con una X grande. Los objetos pueden autodestruirse (como se muestra en la figura 2.15, o pueden ser destruidos mediante otro mensaje (véase la figura 2,16).

Se pueden mostrar con más claridad las consecuencias de la autodelegación cuando se tienen activaciones. Sin ellas, o sin la notación de apilamiento que se usa aquí, es difícil decir dónde ocurrirán más llamadas tras una autodelegación: en el método invocador o en el invocado. Las activaciones de apilamientos dejan en claro este aspecto.

Descubro que, en algunas ocasiones, ésta es una razón para utilizar activaciones en una interacción de procedimiento, aun cuando por lo común no uso las activaciones en estos casos.

Las figuras 2.15 y 2.16 muestran dos de las situaciones del caso de uso de "revisión de transacciones".

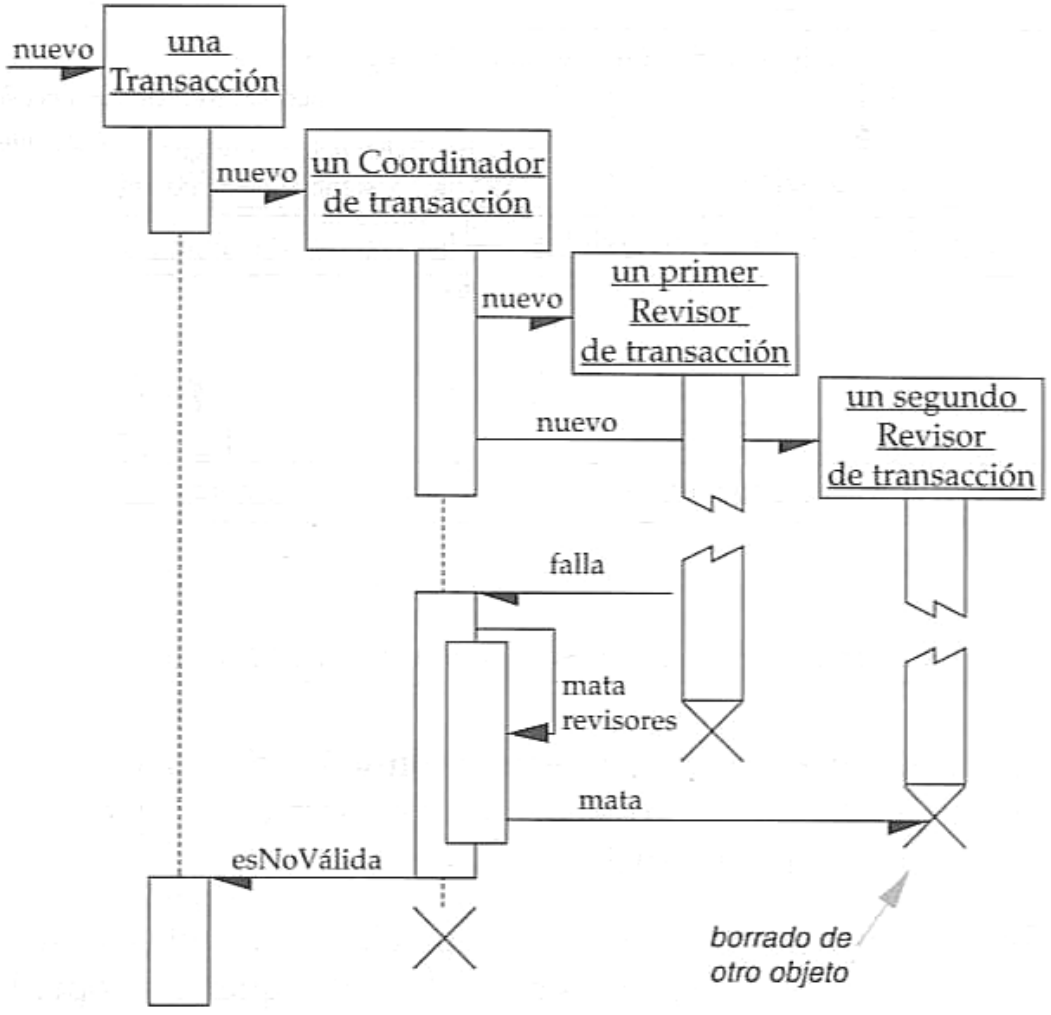


Figura 2.16 diagrama de secuencia 2

En la figura 2.16, una técnica muy útil es insertar descripciones textuales de lo que sucede; en el diagrama de secuencia al lado izquierdo de este. Ello implica la alineación de cada bloque de texto con el mensaje apropiado dentro del diagrama. Esto ayuda a comprender el diagrama se realiza con aquellos documentos que se conservaran, pero no con bosquejos desde cero.

2.2.3 Estados y transición.

Los diagramas de estados son una técnica conocida para describir el comportamiento de un sistema. Describen todos los estados posibles en los que puede entrar un objeto particular y la manera en que cambia el estado del objeto, como resultado de los eventos que llegan a él. En la mayor parte de las técnicas OO, los diagramas de estados se dibujan para una sola clase, mostrando el comportamiento de un solo objeto durante todo su ciclo de vida.

Los diagramas UML de máquina de estados representan los diferentes estados que el objeto puede tener y las transiciones entre los estados.

Un estado representa una etapa en el patrón de comportamiento de un objeto, y es posible tener un estado inicial y un estado final. Un estado inicial, también llamado estado de creación, es cuando un objeto es creado por primera vez, mientras que un estado final es aquel en el que no se llevan a cabo ninguna transición más. Una transición es una progresión de un estado a otro y se activará por un evento ya sea interna o externo al objeto.

El estado identifica un periodo de tiempo del objeto (no instantáneo) en el cual el objeto está esperando alguna operación, tiene cierto estado característico o puede recibir cierto tipo de estímulos. Se representa mediante un rectángulo con los bordes redondeados, que puede tener tres compartimientos: uno para el nombre, otro para el valor característico de los atributos del objeto en ese estado y otro para las acciones que se realizan al entrar, salir o estar en un estado (entry, exit o do, respectivamente).

El evento es una ocurrencia que puede causar la transición de un estado a otro de un objeto. Esta ocurrencia puede ser una de varias cosas:

- Condición que toma el valor de verdadero o falso
- Recepción de una señal de otro objeto en el modelo
- Recepción de un mensaje
- Paso de cierto período de tiempo, después de entrar al estado o de cierta hora y fecha particular

El nombre de un evento tiene alcance dentro del paquete en el cual está definido, no es local a la clase que lo nombre.

Envío de mensajes además de mostrar la transición de estados por medio de eventos, puede representarse el momento en el cual se envían mensajes a otros

objetos. Esto se realiza mediante una línea punteada dirigida al diagrama de estados del objeto receptor del mensaje.

Una transición simple es una relación entre dos estados que indica que un objeto en el primer estado puede entrar al segundo estado y ejecutar ciertas operaciones, cuando un evento ocurre y si ciertas condiciones son satisfechas. Se representa como una línea sólida entre dos estados, que puede venir acompañada de un texto con el siguiente formato:

event-signature "[" guard-condition] "/" action-expression "^"send-clause

event-signature es la descripción del evento que da lugar la transición, **guard-condition** son las condiciones adicionales al evento necesarias para que la transición ocurra, **action-expression** es un mensaje al objeto o a otro objeto que se ejecuta como resultado de la transición y el cambio de estado y **send-clause** son acciones adicionales que se ejecutan con el cambio de estado, por ejemplo, el envío de eventos a otros paquetes o clases.

Una transición interna permanece en el mismo estado, en vez de involucrar dos estados distintos. Representa un evento que no causa cambio de estado. Se denota como una cadena adicional en el compartimiento de acciones del estado.

Las acciones: especifican la solicitud de un servicio a otro objeto como consecuencia de la transición. Se puede especificar el ejecutar una acción como consecuencia de entrar, salir, estar en un estado, o por la ocurrencia de un evento.

Generalización de Estados:

- Podemos reducir la complejidad de estos diagramas usando la generalización de estados.
- Distinguimos así entre superestado y subestados.
- Un estado puede contener varios subestados disjuntos.
- Los subestados heredan las variables de estado y las transiciones externas.
- La agregación de estados es la composición de un estado a partir de varios estados independientes.

La composición es concurrente por lo que el objeto estará en alguno de los estados de cada uno de los subestados concurrentes. La destrucción de un objeto es efectiva cuando el flujo de control del autómata alcanza un estado final no anidado. La llegada a un estado final anidado implica la subida al superestado asociado, no el fin del objeto.

Un estado puede descomponerse en subestados, con transiciones entre ellos y conexiones al nivel superior. Las conexiones se ven al nivel inferior como estados de inicio o fin, los cuales se suponen conectados a las entradas y salidas del nivel inmediatamente superior.

Una transición compleja relaciona tres o más estados en una transición de múltiples fuentes y/o múltiples destinos. Representa la subdivisión en threads del control del objeto o una sincronización. Se representa como una línea vertical de la cual salen o entran varias líneas de transición de estado.

Una transición de hacia un estado complejo (descrito mediante estados anidados) significa la entrada al estado inicial del subdiagrama. Las transiciones que salen del estado complejo se entienden como transiciones desde cada uno de los subestados hacia afuera (a cualquier nivel de profundidad).

Transiciones temporizadas

- Las esperas son actividades que tienen asociada cierta duración.
- La actividad de espera se interrumpe cuando el evento esperado tiene lugar.

Este evento desencadena una transición que permite salir del estado que alberga la actividad de espera. El flujo de control se transmite entonces a otro estado.

PASOS QUE SE SIGUEN PARA SU CONSTRUCCIÓN

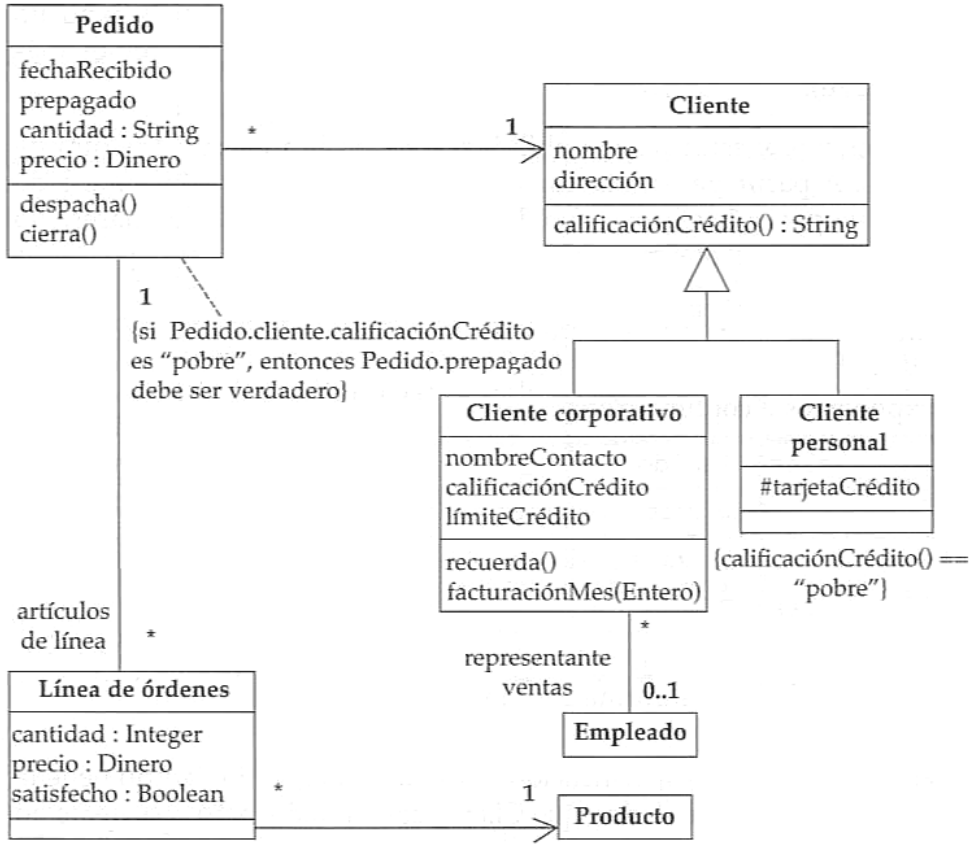


Figura 2.17 modelo del dominio sistema de pedidos

Existen muchas formas de diagramas de estados, cada una con semántica ligeramente diferente. La figura 2.17 muestra el modeolo de domini correpondinte a un sistema de pedidos

La figura 2.18 muestra un diagrama de estados de UML para un pedido del sistema de procesos de pedidos. El diagrama indica los diversos estados de un pedido.

Comenzaremos en el punto de partida y mostramos una transición inicial al estado de Comprobación. Esta transición esta etiquetada como “/obtener el primer artículo”. La sintaxis de una etiqueta de transición tiene tres partes, las cuales son optativas: Evento/[Guard guardia]/Acción. En este caso solo tenemos la acción”obtiene primer artículo” Una vez realizada tal acción, entramos al estado de comprobación. Este estado tiene una actividad asociada con el, la cual se indica mediante una etiqueta con la sintaxis hace/actividad. En este caso, la actividad se llama “comprueba articulo”.

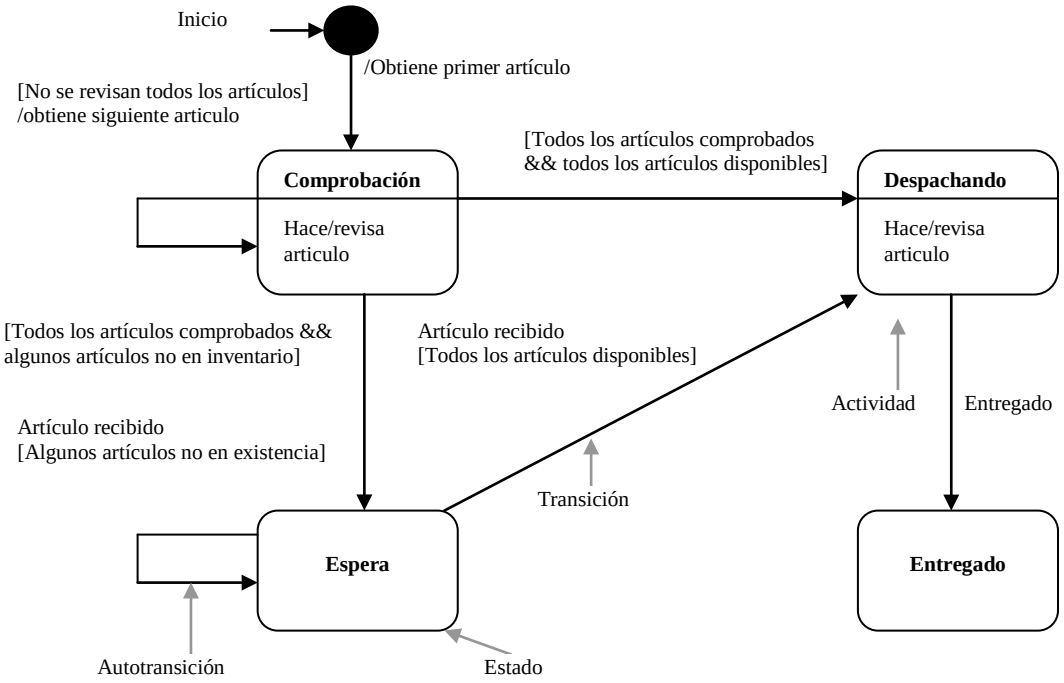


Figura 2.18: Diagrama de Estado

El término “acción” indica la transición, y “actividad” para indica el estado. Las acciones se asocian con las transiciones y se consideran como procesos que suceden con rapidez y no son interrumpibles. Las actividades se asocian con los estados y pueden tardar más. Una actividad puede ser interrumpida por algún evento.

Adviértase que la definición de “rápidamente” depende del tipo de sistema que se está produciendo. En un sistema de tiempo real, “rápidamente” puede significar unas pocas líneas de código de maquina; en los sistemas de información normales, “rápidamente” puede significar menos de unos cuantos segundos.

Cuando una transición no tiene evento alguno en su etiqueta, significa que la transición se da tan pronto como se completa cualquier actividad asociada con el estado dado. En este caso, ello significa tan pronto termine la Comprobación, del estado Comprobación se derivan tres transiciones. Las tres solo tienen guardias en su etiqueta. Un guardia es una condición lógica que solo devuelve “verdadero” o “falso” Una transición de guardia ocurre solo si el guardia se resuelve como “verdadero”.

Solo se puede tomar una transición de un estado dado, por lo que los guardias sean mutuamente excluyentes para cualquier evento. En la figura 2.18

1. si no hemos comprobado todos los artículos, tomamos el siguiente artículo y regresamos al estado de comprobación para comprobarlo.
2. Si hemos comprobado todos los artículos y todos están en existencia, hacemos la transición al estado de Despachando.
3. si hemos comprobado todos los artículos pero no todos están en existencia, hacemos la transición al estado espera.

En primer lugar, el estado Espera, como no hay actividades para este estado, el pedido se detiene en el esperando un evento, ambas transiciones del estado espera se etiquetan con el evento Articulo recibido.

Dentro del estado despachando, la actividad que inicia una entrega. También hay una transición simple no guardada, disparada por el evento Entregado. Esto significa que la transición ocurrirá siempre que tenga lugar el evento. Sin embargo, nótese que la transición no sucede cuando termina la actividad; por el contrario, una vez terminada la actividad “iniciar entrega”, el pedido se mantiene en el estado Despachando, hasta que ocurre el evento Entregado

La transición denominada “cancelado”. Se tiene la posibilidad de cancelar un pedido en cualquier momento, antes de que sea entregado.

Las figuras 2.19 y 2.20 muestran como estos enfoques reflejan e mismo comportamiento del sistema.

En los ejemplos actuales, se ha mostrado una actividad dentro de un estado, indicándola con texto en la forma de hace/actividad. También se puede indicar otras cosas en un estado.

Si un estado responde a un evento con una acción que no produzca una transición, dicha condición se puede mostrar poniendo un texto de la forma nombreEvento/nombreAcción en el cuadro de estado

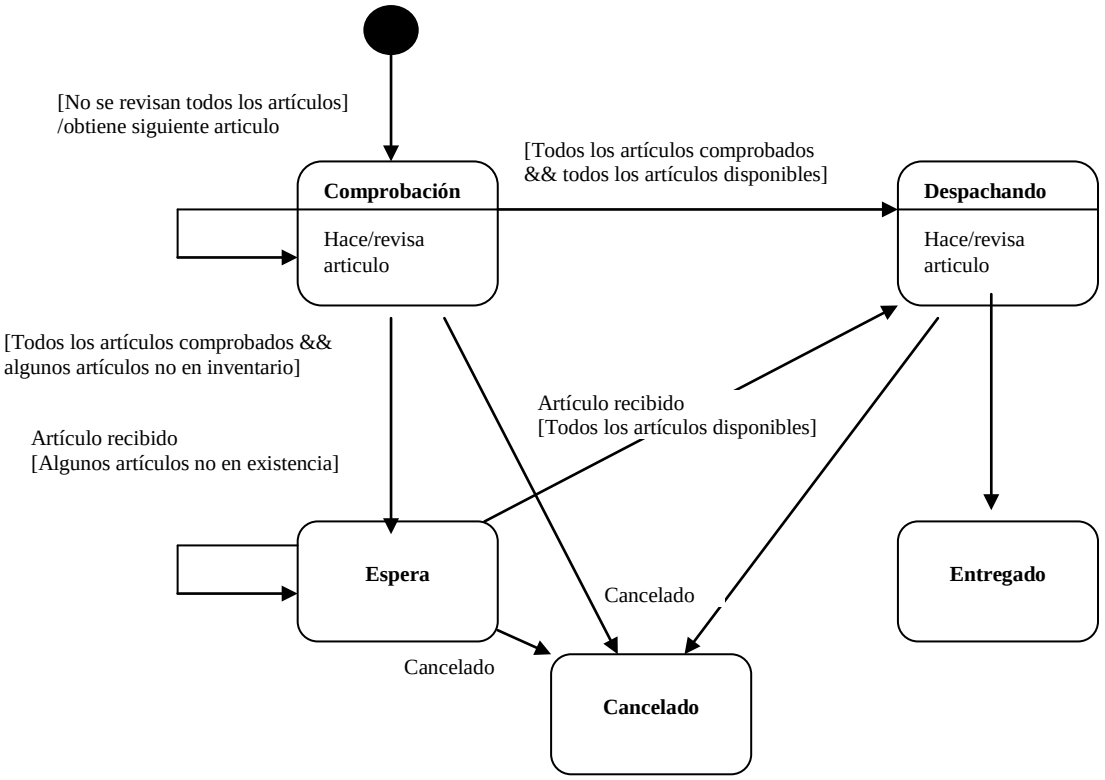


Figura 2.19: Diagrama de Estado sin superestado

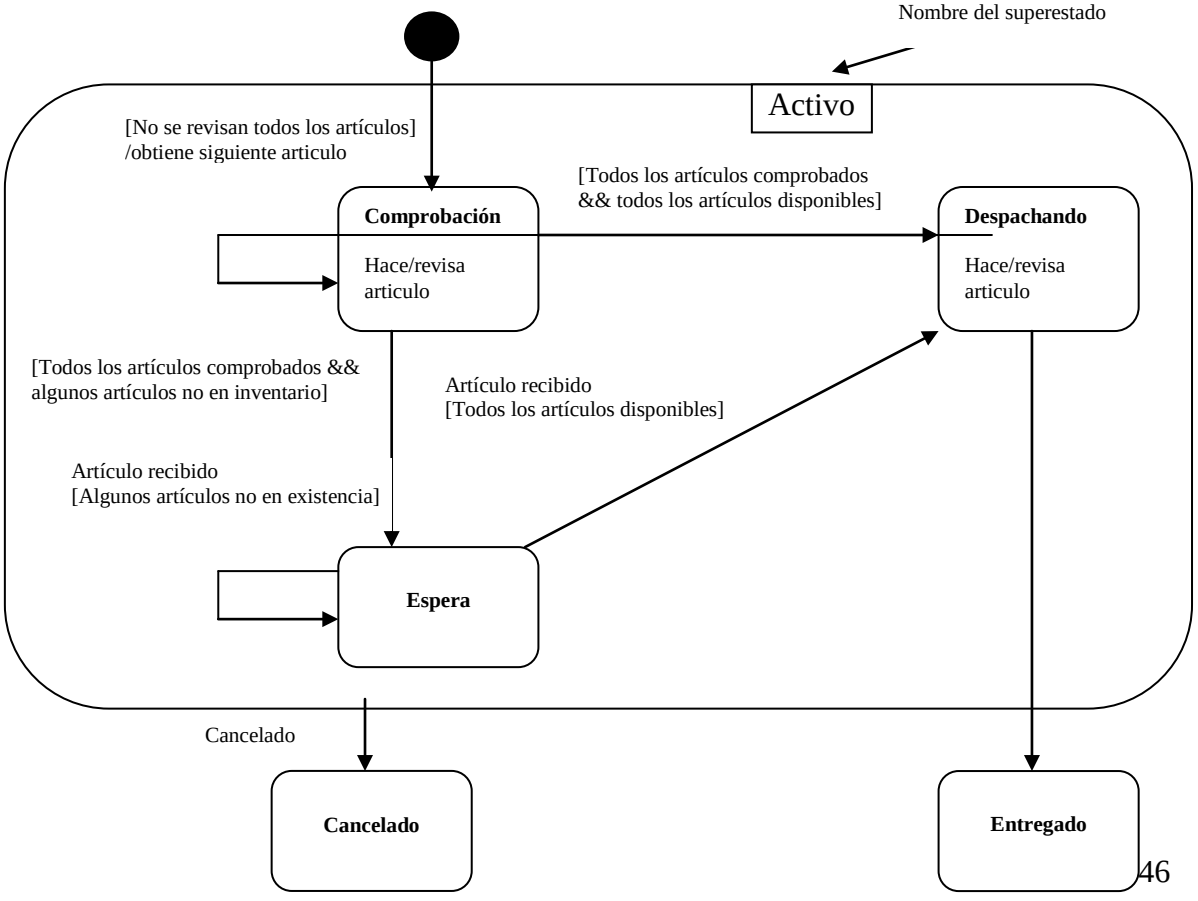


Figura 2.20: Diagrama de Estado con superestado

Existen también dos eventos especiales, entrada y salida. Cualquier acción que este marcada como vinculada al evento entrada se ejecuta siempre que se entre al estado dado a través de una transición. La acción asociada con el evento salida se ejecuta siempre que se sale del estado por medio de una transición. Si se tiene una transición que vuelve al mismo estado (a esto se llama auto transición) por medio de una acción, se ejecuta primero la acción de salida, luego la acción de transición y, por último la acción de entrada. Si el estado tiene también una actividad asociada esta se ejecuta tras la acción de entrada.

4.1 Diagramas de estados concurrentes

Además de los estados de un pedido que se basan en la disponibilidad de los artículos, existen estados que se basan en la autorización de pagos. Si vemos estos estados, podríamos ver un diagrama de estados como el de la figura 2.21

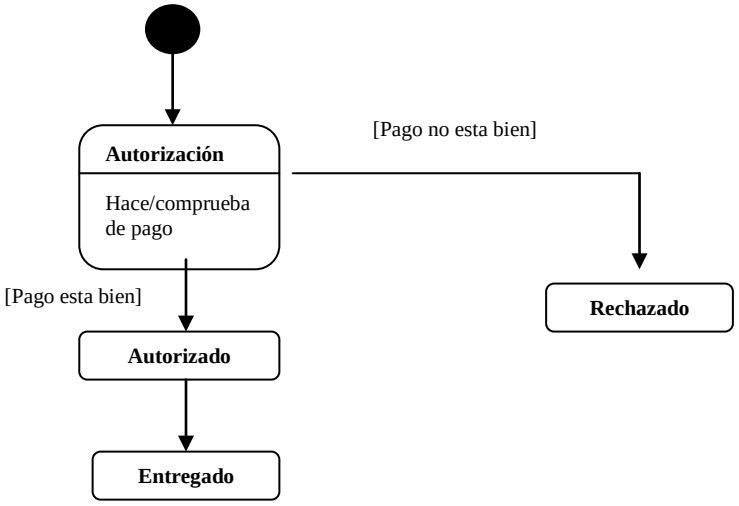


Figura 2.21: Autorización de Pagos

Aquí comenzamos efectuando una autorización. La actividad de “comprobación de pago” termina señalando que el pago fue aprobado. Si el pago esta bien, el pedido espera en el estado autorizado hasta que sucede el evento de “entrega”. De otro modo, el pedido entra al estado de Rechazado.

El objeto Orden presenta una combinación de los comportamientos que se muestran en las figuras 2.19 y 2.20. Los estados asociados y el estado Cancelado mencionado anteriormente pueden combinarse en un diagrama de estados concurrentes figura 2.22.

Nótese que en la figura 2.22 se ha dejado fuera los detalles de los estados internos.

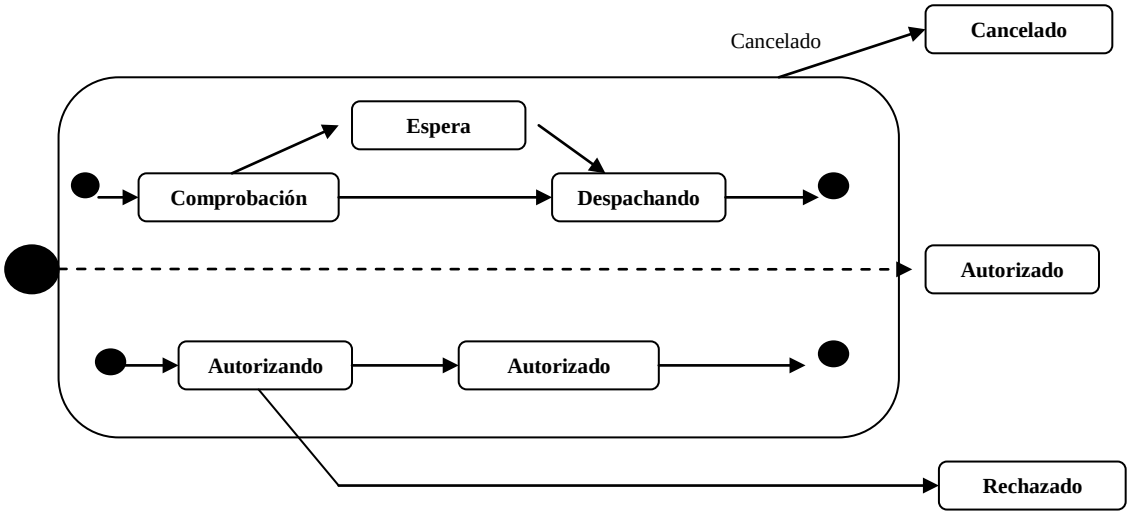


Figura 2.22: Diagrama de estados concurrentes

Los diagramas de estados concurrentes son útiles cuando un objeto dado tiene conjuntos de comportamientos independientes. Nótese, sin embargo, que no se debe permitir que sucedan demasiados conjuntos de comportamientos concurrentes en un solo objeto. Si se tienen varios diagramas de estados concurrentes complicados para un solo objeto, se deberá considerar la división del objeto en varios

Los diagramas de estados son buenos para describir el comportamiento de un objeto a través de varios casos de uso. No son tan buenos para describir un comportamiento que involucra cierto número de objetos que colaboran entre ellos. Así pues, es útil combinar los diagramas de interacción son buenos para la descripción del comportamiento de varios objetos en un mismo caso de uso. Por su parte, los diagramas de actividades son buenos para mostrar la secuencia general de las acciones de varios objetos y casos de uso.

Si decide utilizar diagramas de estado, no trate de dibujar uno por cada clase del sistema. Aunque este es el enfoque que emplean los detallistas ceremoniosos, casi siempre es un esfuerzo inútil. Utilice los diagramas de estados solo para aquellas clases que presenten un comportamiento interesante, cuando la construcción de tales diagramas le ayude a comprender lo que sucede. Muchos consideran que los objetos de interfaz de usuario (IU) y de control tienen el tipo de comportamiento que es útil describir mediante diagramas de estado

3. Construcción.

El análisis da como resultado el modelo de requerimientos descrito por los siguientes productos:

- Un conjunto de requisitos no funcionales y las limitaciones, tales como el tiempo máximo de respuesta, rendimiento mínimo, confiabilidad, plataforma de sistema operativo, y así sucesivamente
- Un modelo de casos de uso, que describe la funcionalidad del sistema desde el punto de vista de los actores
- Un modelo de objetos, que describe las entidades manipuladas por el sistema
- Un diagrama de secuencia para cada caso de uso, que muestra la secuencia de interacciones entre los objetos que participan en el caso de uso

El modelo de análisis describe el sistema completo desde el punto de los actores de vista y sirve como base de la comunicación entre el cliente y los desarrolladores. El modelo de análisis, sin embargo, no contiene información sobre la estructura interna del sistema, su configuración de hardware o, en términos generales, la manera en que se debe realizar. El diseño del sistema es el primero paso en esta dirección. El diseño del sistema da como resultado los siguientes productos:

- Una lista de objetivos de diseño, que describe las cualidades del sistema que los desarrolladores deben optimizar
- Una arquitectura de software, que describe la descomposición del subsistema en subsistemas desde el punto de vista de responsabilidades del subsistema, dependencias entre subsistemas, correspondencias de los subsistema con el hardware, y decisiones políticas principales, como el flujo de control, control de acceso y almacenamiento de datos.

Los objetivos de diseño se derivan de los requisitos no funcionales. Los objetivos de diseño guían las decisiones que deben tomar los desarrolladores, en especial cuando hay compromisos. La descomposición en subsistemas constituye la mayor parte del diseño del sistema. Los desarrolladores dividen el sistema en piezas manejables para hacer frente a la complejidad: Cada subsistema se asigna a un equipo y se realiza en forma independiente. Sin embargo para que esto sea posible, los desarrolladores necesitan atacar asuntos en el nivel de sistema cuando descomponen el sistema. En particular, necesitan atacar las siguientes cuestiones:

- Correspondencia entre Hardware y software: ¿Cuál es la configuración de hardware del sistema? ¿Cuál nodo es el responsable de que funcionalidad? ¿Cómo se realiza la comunicación entre los nodos? ¿Cuáles servicios se realizan utilizando componentes de software existentes? ¿Cómo se encapsulan estos componentes?

- Administración de datos: ¿Cuáles datos deben ser persistentes? ¿Dónde se almacenan los datos persistentes? ¿Cómo se accede a ellos?
- Control de acceso: ¿Quién puede acceder a los datos? ¿Pueden cambiar manera dinámicamente el control de acceso? ¿Cómo se especifica y realiza el control de acceso? El control de acceso y la seguridad son asuntos en el nivel del sistema. El control de acceso debe ser consistente en todo el sistema, es decir, la política se utiliza para especificar quién puede y quién no puede acceder a ciertos datos debe ser la misma en todos los subsistemas.
- Control de flujo: ¿Cómo funciona la secuencia de las operaciones del sistema? El sistema es manejado por eventos? ¿Puede manejar más de una interacción del usuario a la vez?
- Condiciones de frontera: ¿Cómo se inicia el sistema? ¿Cómo se apaga? ¿Cómo se detectan y manejan los casos de excepción?

3.1 Despliegue de componentes y arquitectónico.

Los grandes sistemas siempre se descomponen en subsistemas que proporcionan algún conjunto de servicios relacionados. El proceso de diseño inicial que identifica estos subsistemas y establece un marco para el control y comunicación de los subsistemas se llama diseño arquitectónico.

El resultado de este proceso de diseño es una descripción de la arquitectura del software.

El proceso de diseño arquitectónico está relacionado con el establecimiento de un marco estructural básico que identifica los principales componentes de un sistema y las comunicaciones entre estos componentes

Con el fin de reducir la complejidad del dominio de aplicación, se identificaron partes más pequeñas llamadas clases y las organizó en paquetes. Del mismo modo, para reducir la complejidad del dominio de solución, se descomponen un sistema en partes más simples, llamados subsistemas, que están constituidos por una serie de clases de del dominio solución. En el caso de subsistemas complejos aplicamos de forma reiterada este principio y se descomponemos en subsistema más simples un subsistema, ver Figura 3.1.

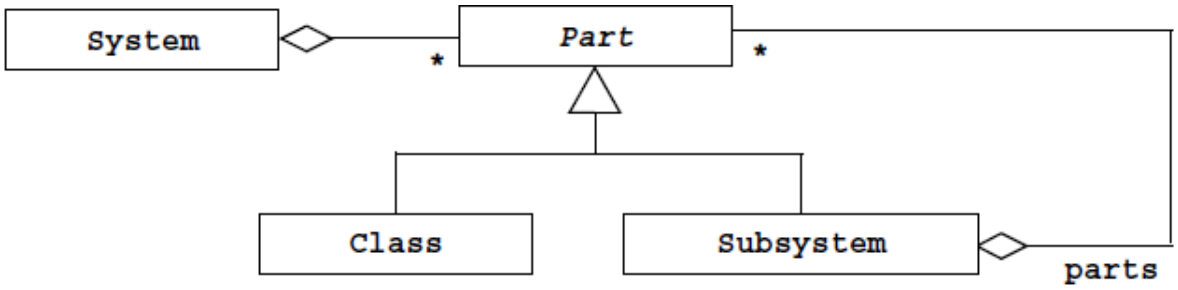


Figura 3.1 diagrama de clases

Ejemplo: en un sistema de gestión de accidentes los oficiales de campo, como por ejemplo, un oficial de policía o bombero, tiene acceso a una computadora inalámbrica que les permite interactuar con un despachador. El despachador, a su vez puede visualizar el estado actual de todos sus recursos, como por ejemplo, coches de policía o camiones, en una pantalla de computadora y despachar un recurso mediante la emisión de comandos desde una estación de trabajo. En este ejemplo, FieldOfficer y Dispatcher son actores.

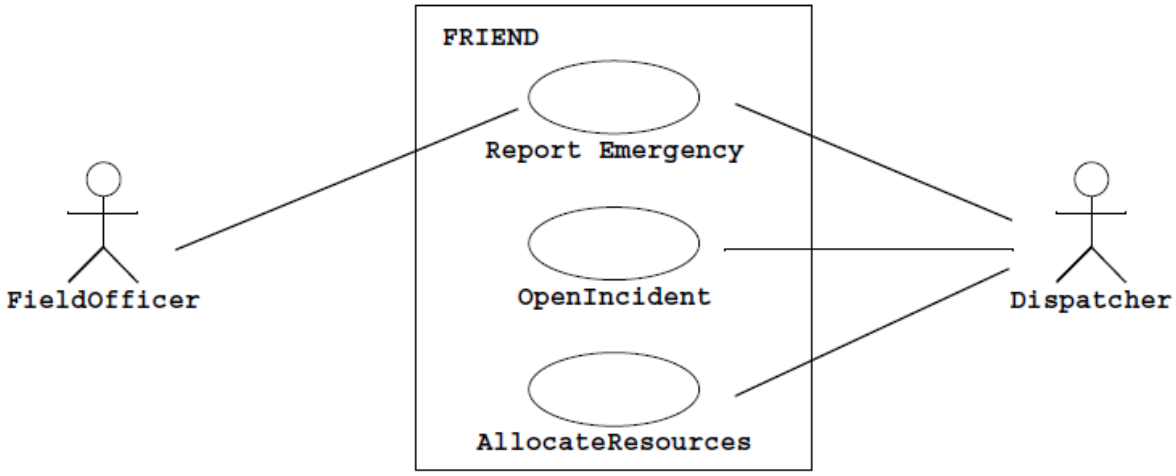


Figura 3.2 Ejemplo de un caso de uso: ReportEmergency

El FieldOfficer activa la función "Reporte de emergencia" de su terminal. Del sistema FRIEND quien responde presentando un formulario al oficial.

El FieldOfficer llena el formulario seleccionando el nivel de emergencia, tipo, ubicación, y una breve descripción de la situación. El FieldOfficer también describe las posibles respuestas a la situación de emergencia. Una vez completado el formulario, el FieldOfficer lo envía y en ese momento se le notifica al Dispatcher.

El Dispatcher revisa la información presentada y crea una Incidente en la base de datos mediante la invocación del caso de uso OpenIncident. El Dispatcher selecciona una respuesta y da un acuse de recibo del reporte de emergencia.

El FieldOfficer recibe el acuse de recibo y la respuesta seleccionada.

Se da acuse de recibo del reporte del Field Officer en menos de 30 segundos. La respuesta seleccionada llega antes de 30 segundos a partir de que la envía el Dispatcher.

Por ejemplo, el sistema de gestión de accidentes anteriormente descrito se puede descomponer en un subsistema DispatcherInterface, la aplicación de la interfaz de usuario para el despachador, un subsistema FieldOfficerInterface, que implementa la interfaz de usuario para el FieldOfficer; un subsistema IncidentManagement, que implementa la creación, modificación y almacenamiento de Incidente, y un subsistema Notificacion, que implementa la comunicación entre terminales

FieldOfficer y las estaciones Dispatcher. Esta descomposición del subsistema se muestra en la Figura 3.4 con los paquetes de UML.

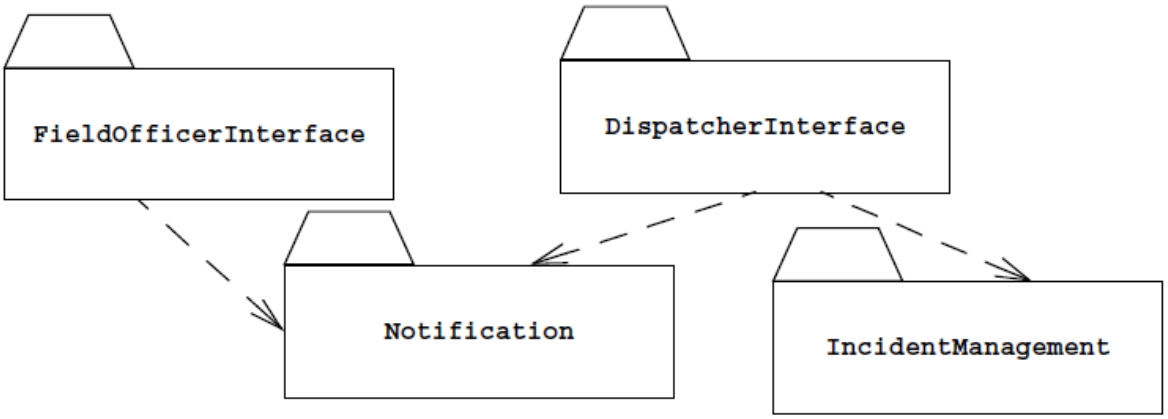


Figura 3.3 Descomposición en subsistemas

Figura 3.3 Descomposición en subsistemas para un sistema de gestión de accidentes. Los subsistemas se muestran como los paquetes de UML. Las flechas de puntos indican las dependencias entre subsistemas.

Un subsistema se caracteriza por los servicios que presta a otros subsistemas. Un servicio es un conjunto de operaciones relacionadas que comparten un propósito común. Un Subsistema proporciona un servicio de notificación, por ejemplo, define las operaciones para enviar avisos, buscar canales de notificación, y suscribirse y cancelar la suscripción a un canal.

Por ejemplo, considere un sistema de seguimiento de decisión para el registro de problemas de diseño, discusiones, evaluaciones alternas, decisiones y su implementación desde el punto de vista de tareas, vea la Figura 3.4.

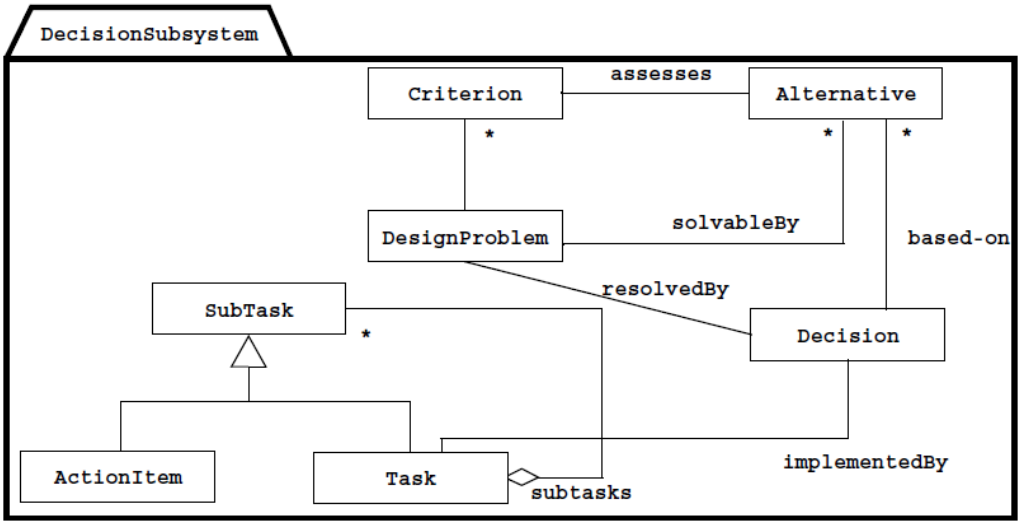


Figura 3.4 Sistema de seguimiento de decisiones.

La Figura 3.4 Sistema de seguimiento de decisiones. El DecisionSubsystem tiene una baja coherencia: Las clases Criterion, Alternative, y DesignProblem no tienen relaciones con Subtask, ActionItem, y Task

El sistema de seguimiento de decisiones es lo suficientemente pequeño que podríamos agrupar a todos estas clases en un subsistema llamado DecisionSubsystem ver Figura 3.4. Sin embargo, se observa que los modelo de clase se puede dividir en dos subgrafos. Uno, llamado RationaleSubsystem, contiene las clases DesignProblem, Alternative, Criterio y Decision. El otro, llamado PlanningSubsystem, contiene Task, Subtask y ActionItem, ver Figura 3.5. Ambos subsistemas tienen una mayor coherencia que el original DecisionSubsystem.

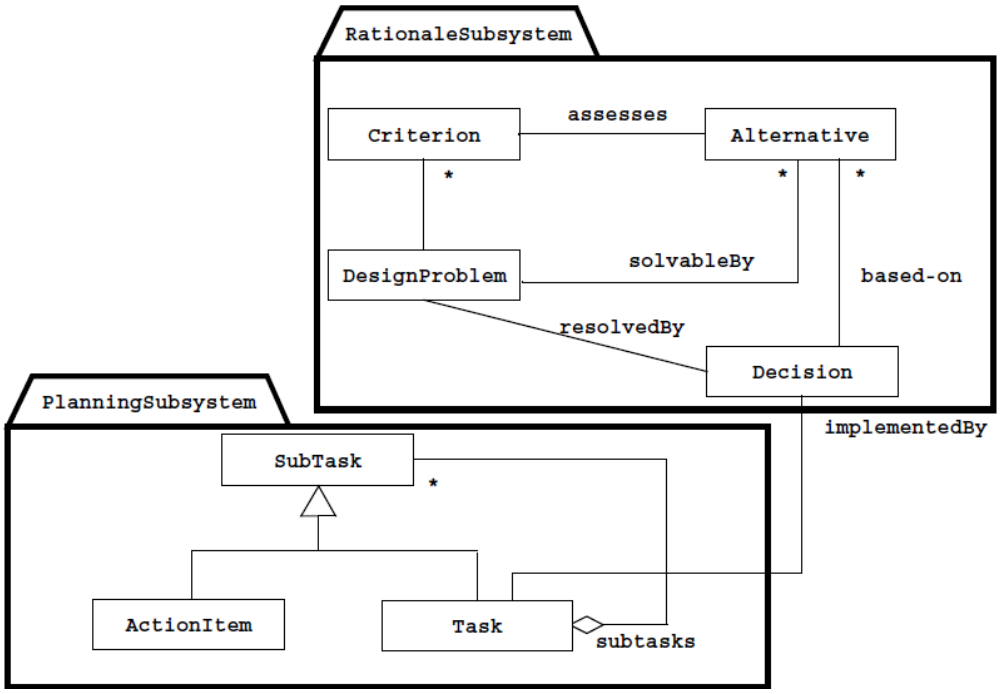


Figura 3.5 alternativas de descomposición

La figura 3.5 muestra alternativas de descomposición del subsistema para el sistema de seguimiento de decisiones. La coherencia del RationaleSubsystem y PlanningSubsystem es superior a la coherencia del original DecisionSubsystem. Note también que se reduce la complejidad por la descomposición del sistema en subsistemas más pequeños.

3.2 Técnicas de desarrollo de las arquitecturas de referencia en diferentes dominios.

Los grandes sistemas siempre se descomponen en subsistemas que proporcionan algún conjunto de servicios relacionados. El proceso de diseño inicial que identifica estos subsistemas y establece un marco para el control y comunicación de los subsistemas se llama diseño arquitectónico. El resultado de este proceso de diseño es una descripción de la arquitectura del software.

El diseño arquitectónico es la primera etapa en el proceso de diseño y representa un enlace crítico entre los procesos de ingeniería de diseño y de requerimientos. El proceso de diseño arquitectónico está relacionado con el establecimiento de un marco estructural básico que identifica los principales componentes de un sistema y las comunicaciones entre estos componentes

El resultado del proceso de diseño arquitectónico es un documento de diseño arquitectónico. Éste puede incluir varias representaciones gráficas del sistema junto con texto descriptivo asociado. Debería describir cómo se estructura el sistema en subsistemas, la aproximación adoptada y cómo se estructura cada subsistema en módulos. Los modelos gráficos del sistema presentan diferentes perspectivas de la arquitectura. Los modelos arquitectónicos que pueden desarrollarse pueden incluir:

a) Un modelo estructural estático que muestre los subsistemas o componentes que han sido desarrollados como unidades separadas.
b) Un modelo de proceso dinámico que muestre cómo se organiza el sistema en procesos en tiempo de ejecución. Este modelo puede ser diferente del modelo estático.
c) Un modelo de interfaz: que defina los servicios ofrecidos por cada subsistema a través de su interfaz pública.
d) Modelos de relaciones que muestren las relaciones. tales como el flujo de datos, entre los subsistemas.
e) Un modelo de distribución, que muestre cómo se distribuyen los subsistemas entre las computadoras.

Los modelos informales y las notaciones tales como UML son las notaciones comúnmente usadas para la descripción arquitectónica.

3.2.1 Los modelos de componentes

De acuerdo con el UML, un componente es un módulo de software ejecutable con su propia identidad e interfaces definidas, es decir, que se define en términos muy generales y abstractos. En relación con las tecnologías de componentes populares, tales como Enterprise Java Beans, donde la definición es muy concreta y tecnológicamente específica.

Un componente consiste en una o más clases y tiene interfaces definidas. Una instancia de componente, como un objeto, tiene su propia identidad. La diferencia

más importante entre los componentes y las clases/objetos, es que los componentes deben ser intercambiables.

Para alcanzar este objetivo, sin embargo, deben cumplirse diversas condiciones:

- Las interfaces deben ser técnicamente
- Las interfaces deben ser compatibles a nivel de dominio específico
- Las dependencias entre los distintos componentes se deben mantener al mínimo.

Esto tiene las siguientes consecuencias para los componentes de software:

- Las interfaces de un componente, en la medida de lo posible, sólo contienen tipos primitivos (Integer, String, etc).
- Las normas apropiadas del dominio específico son necesarios para la compatibilidad del dominio específico.
- Las asociaciones entre los componentes, en la medida de lo posible, se sustituirá por mensajes y mecanismos de observación.

Los componentes por lo tanto suelen tener tres tipos diferentes de interfaz (Figura. 3.6):

- una interfaz para la generación o la carga de instancias, por ejemplo, con la ayuda de una clave específica del dominio. En EJB esto normalmente sería llamado BuscarPorClave(),
- una interfaz que define la observación y servicios de mensajes,
- un dominio específico de interfaz (interfaz de objeto).

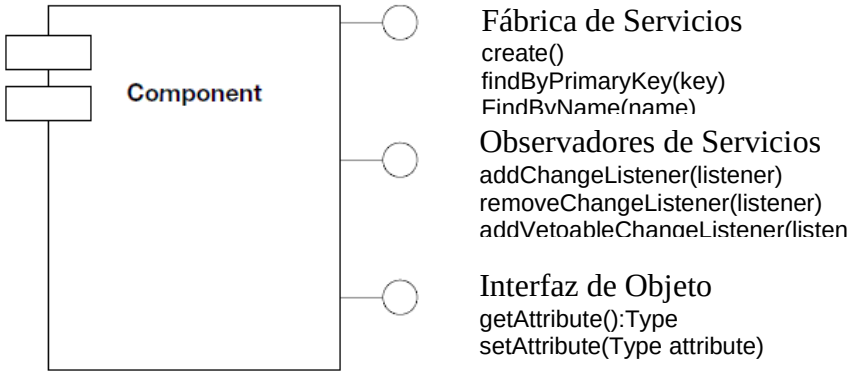


Figura 3.6 interfaces principales de un componente

Independiente de los valores de sus atributos, los objetos tienen una identidad única, que dependen en general de las asociaciones. A fin de minimizar el

resultado de las dependencias entre los componentes, las relaciones pueden llevarse a cabo entre las instancias del componente, por ejemplo, mientras que sólo la clave específica de dominio se almacena como la referencia. La comunicación entre los componentes no puede ser abordado directamente, sino que primero debe ser explícitamente cargados por medio de la clave de dominio específico.

Por otra parte, el intercambio de mensajes se lleva a cabo a nivel anónimo con la ayuda de los mecanismos de observación llamada. Este tipo de funciones de comunicación en la forma que una instancia remite a los otros el mensaje de addListener (), que significa algo así como "por favor, me informan si ha cambiado.

Mensajes de modificación. Tan pronto como la instancia cambia, se envía un mensaje a todos los suscriptores diciéndoles que ha cambiado.

Antes de comenzar las actividades de diseño, es necesario especificar la arquitectura de las aplicaciones futuras. Determinar qué tipo de clases y, que interfaces deben ser diseñadas. Una bien pensada arquitectura de la aplicación ayuda a lograr:

- una división racional de trabajo y una visión clara;
- flexibilidad a largo plazo en el desarrollo del sistema;
- un mayor grado de aptitud para la reutilización

Un componente de dominio generalmente se compone internamente de un conjunto de clases de entidad/control, y es persistente.

Durante el análisis del siguiente ejemplo, empresa de alquiler, los procesos de negocio, que se identifica son: administración de vehículos, alquiler de vehículos y atención al cliente de la Figura 3.7. Estos ahora cada uno será representado por un componente en la Figura. 3.8.

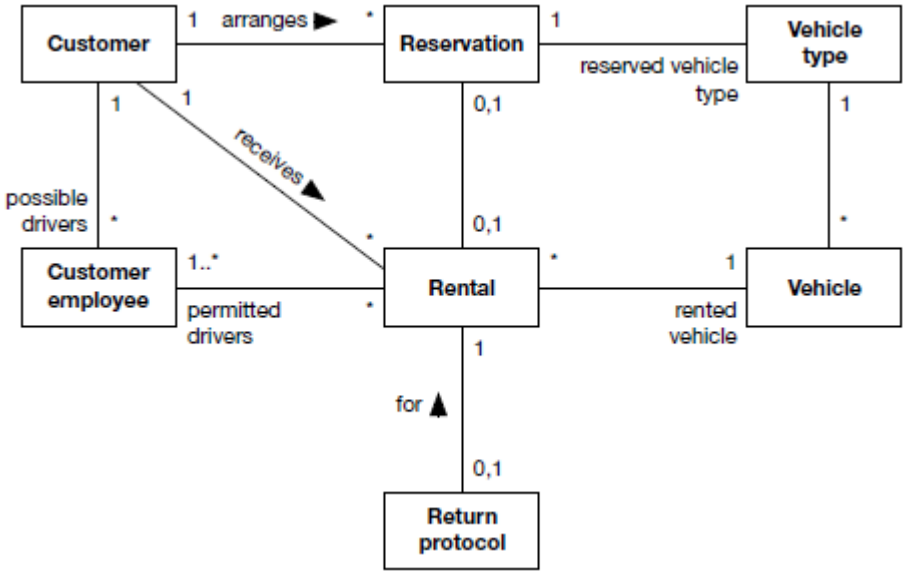


Figura 3.7 Las clases más importantes de dominio específico y relaciones representadas como un modelo de análisis de clase del sistema alquiler de vehículos

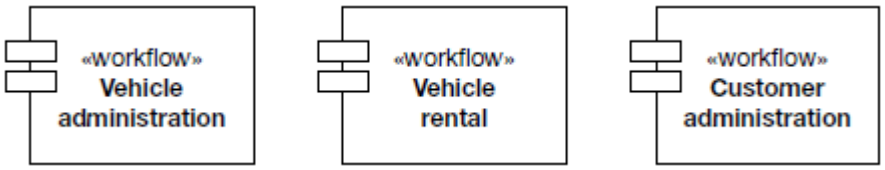


Figura 3.8 Los componentes del flujo de trabajo en el ejemplo

Definir componente controlador del caso de uso para cada caso de uso. En este sentido, comenzar con los casos de uso del sistema que se identificaron en el análisis, por ejemplo, Vehículo de reserva por teléfono ver figura. 3.10 correspondiente al caso de uso del mismo nombre de la figura 39.

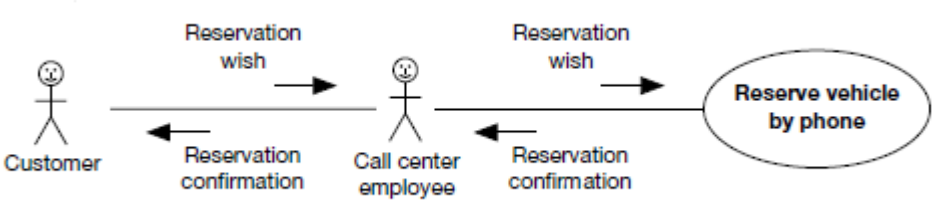


Figura 3.9 El sistema de casos concretos de reservación de vehículo

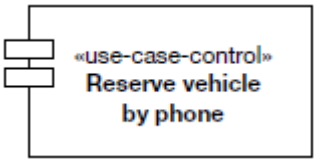


Figura 3.10 El controlador caso de uso reserva de vehículo por teléfono como un componente de control

Representan a cada sistema externo de un componente. Las interacciones principales del sistema con el mundo exterior, los actores, se muestran en los modelos de sistemas de casos de uso. Los agentes también pueden ser sistemas heredados, externos o los llamados a estos. Para llevar a cabo la interacción con estos sistemas externos, la arquitectura de la aplicación específica, establece que deben ser encapsulados como componentes.

Se combinan las clases relacionadas en el modelo de análisis para formar los componentes de dominio. Las clases que figuran en el modelo de clases de

análisis representan el punto de partida más importante para determinar los componentes de dominio como se muestra en la figura 3.11.

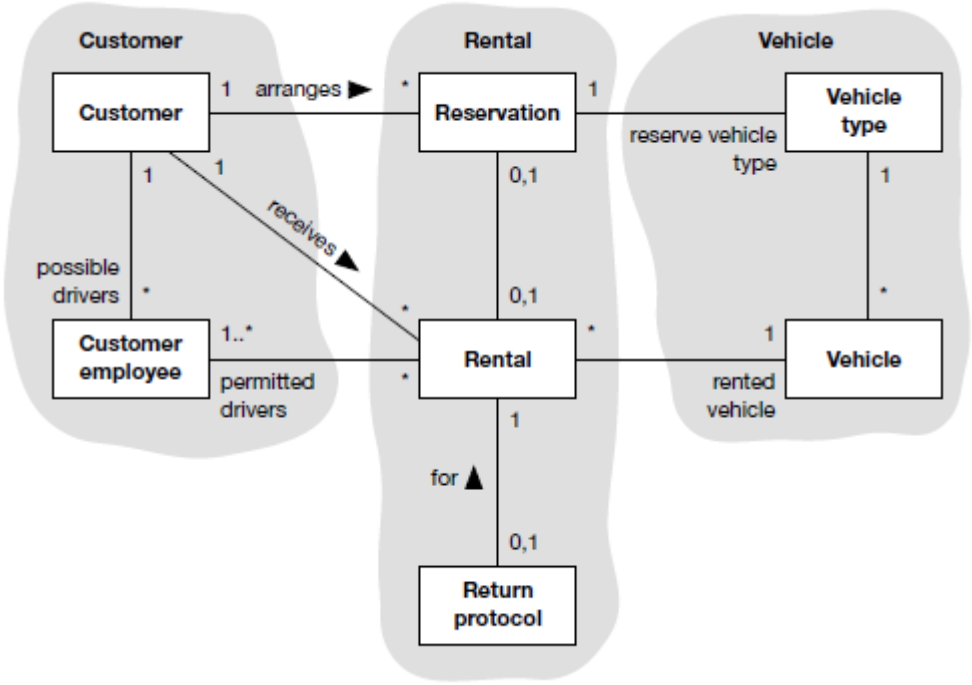


Figura 3.11 División de las clases en el modelo de clase de análisis entre los componentes planeados

Estimar cuán grande son las dependencias entre las clases y cuán estrechamente relacionados se encuentran en un nivel de dominio específico, entonces agruparlas según corresponda para formar componentes. En nuestro ejemplo ver figura. 3.11 esto produce los componentes Customer, Rental y de vehícle. Como la figura. 3.12 muestra, que no existe hasta ahora ninguna dependencia conocida entre el Customer y de vehicle.

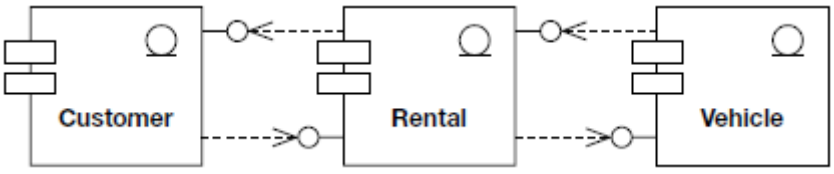


Figura 3.12 Un primer componente del modelo de dominio.

En resumen:

- Basado en el modelo de clase del análisis, desarrollar un modelo de diseño clase específica para cada componente.
- Utilizare las claves de un dominio específico para representar a todas las asociaciones a los objetos de otros componentes.

- Definir las responsabilidades de todas las clases de componentes específicos, es decir, transformar la descripción del problema (análisis) en un concepto de la solución (diseño) y si es necesario reestructurar los modelos de clase de componente específico.

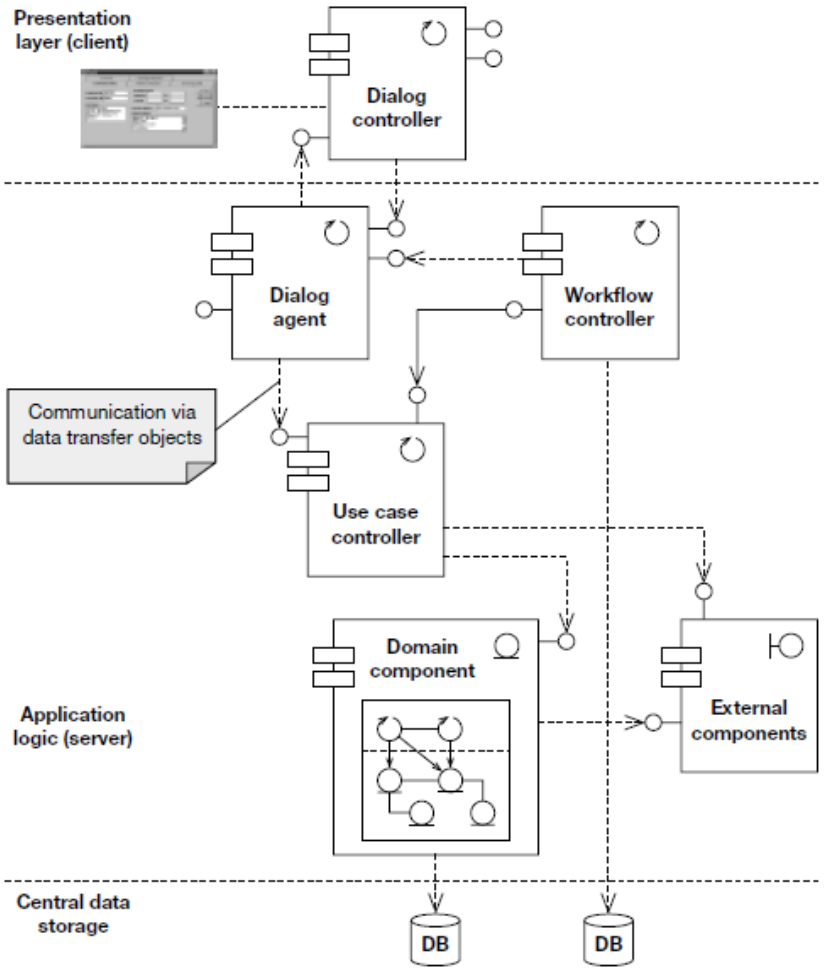


Figura 3.13 La arquitectura de la aplicación asumida para el ejemplo

3.2.2 Arquitectura de referencia para sistemas de tiempo real fuente de alimentación

En su forma más simple, un sistema informático consta de una CPU y la memoria interconectados por un bus como se muestra en la figura 3.14.

Hay tres buses en todo el sistema: el de corriente eléctrica, direcciones y datos.

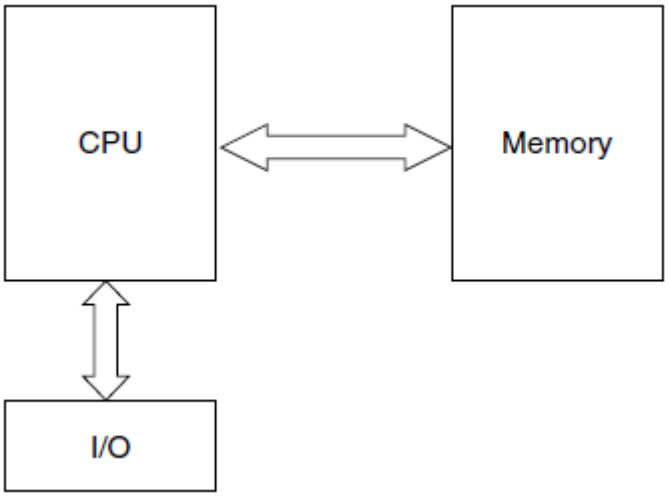


Figura 3.14 Arquitectura Von Newman

Al crear sistemas en tiempo real empotrados no hay margen para el error. La naturaleza demanda de un producto final que será poderoso, eficiente y confiable.

Los desarrolladores sofisticados confían en las soluciones de patrones de diseño —que proveen soluciones a cambios recurrentes de diseño— para la construcción de sistemas de evaluación en tiempo real y a prueba de fallos. Los patrones de diseño en tiempo real es la referencia más importante para los desarrolladores que buscan emplear esta técnica de gran alcance.

El UML es ampliamente aceptado como el lenguaje estándar de facto para la especificación y diseño de sistemas intensivos en software utilizando un enfoque orientado a objetos. Al reunir a los "mejores de su clase" en técnicas de especificación.

Con respecto a los sistemas en tiempo real son los modelos de comportamiento UML los que son de interés.

Un Sistema de Tiempo Real (STR) puede definirse como aquél que debe completar sus actividades en plazos de tiempo predeterminados.

Un campo clásico de aplicación de los Sistemas de Tiempo Real es el de los sistemas de control. No obstante, muchos sistemas de tiempo real no pertenecen a los denominados sistemas de control.

Un ejemplo típico de un sistema de control es un controlador remoto de la temperatura de un horno de fundición de aceros especiales que actúa en función de la temperatura del mismo.

El sistema de control (controlador) puede leer los sensores de temperatura del horno cada cierto tiempo, éste en función de los valores recibidos, del instante de

tiempo considerado y de la situación del horno decide incrementar o decrementar la temperatura actuando sobre el sistema de calentamiento.

Para controlar o monitorizar un sistema externo, el diseñador del STR deberá construir un modelo del sistema a controlar en el que incluya la información necesaria para poder controlar la evolución dinámica del mismo. A esta información se le denomina **estado**.

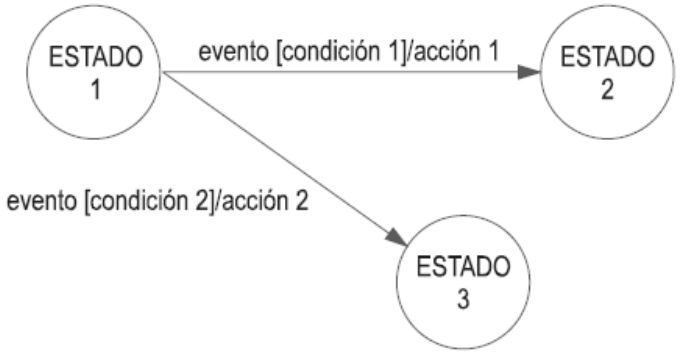


Figura 3.15. Diagrama de estados

Un mismo evento puede dar lugar a dos estados distintos en función de alguna condición (relacionada con algún valor de una variable o del reloj). El cambio de estado puede provocar otro evento o acción sobre otra parte del sistema.

El desarrollo de sistemas de tiempo real podría hacerse como cualquier otro sistema de software empleando cualquier modelo de ciclo de vida. No obstante, la importancia que los aspectos temporales tienen en este tipo de sistemas hace que la atención del diseñador deba concentrarse en algunos aspectos desapercibidos en sistemas de información [

En un sistema de computador embebido generalizado cada computador interactúa directamente con el equipamiento físico en el mundo real. Con el fin de controlar esos dispositivos del mundo real, el computador necesitará muestrear los dispositivos de medida a intervalos regulares; por lo tanto, se precisa un reloj de tiempo real.

3.2.3 Arquitectura de referencia para sistemas móviles con conexión a Internet.

La arquitectura básica de una aplicación web incluye navegadores, una red y un servidor web Figura 3.16. Navegadores solicitan "páginas web" desde el servidor. Cada página es una mezcla de contenido y el formato de instrucciones, expresada con HTML. Algunas páginas incluyen secuencias de comandos de cliente que son interpretados por el navegador. Estas secuencias de comandos definen

comportamiento dinámico adicional para la página de presentación y a menudo interactúan con el navegador, páginas de contenido y controles adicionales (Applets, controles ActiveX y plug-ins). El usuario ve e interactúa con el contenido de la página. A veces el usuario introduce información en elementos de campo de la página y la entrega al servidor para su procesamiento. El usuario también puede interactuar con sistema desplazándose a diferentes páginas del sistema a través de hipervínculos. En cualquier caso, el usuario es el suministro de entrada al sistema que puede alterar el "Estado del negocio" del sistema.

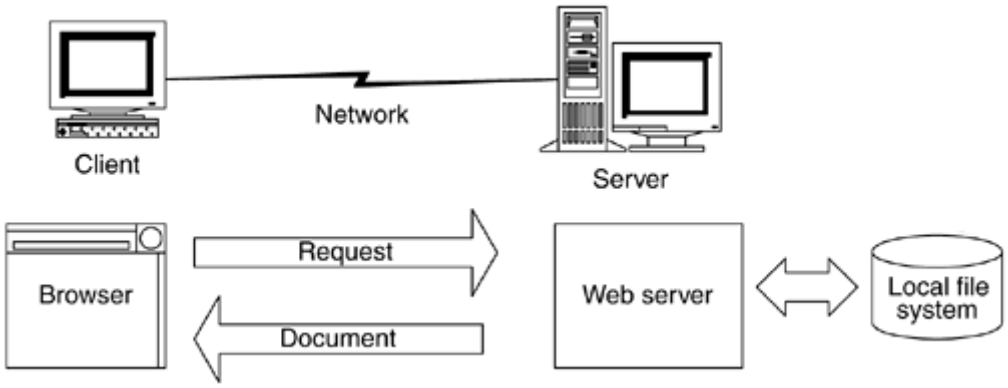


Figura 3.16. Sistema Web Básico

Los servidores web de hoy han mejorado el diseño básico. Hoy están mucho más conscientes de la seguridad e incluir características como administración de estado de cliente en el servidor, integración de procesamiento de transacción, administración remota y recursos agrupación para mencionar sólo algunas.

Los servidores web de hoy en día se pueden dividir en tres categorías principales: secuencias de comandos, páginas y páginas compiladas un híbrido de los dos.

Modelando páginas Web

Páginas, Web ya sea mediante secuencias de comandos o compilado, mapea uno a uno a los componentes de UML.

Un componente es una parte "física" y reemplazable del sistema. La vista de aplicación (vista de componentes) del modelo describe los componentes del sistema y sus relaciones. En un nivel, un diagrama de componentes de un sistema web es como un mapa del sitio.

Como los componentes representan sólo la presentación física de interfaces, no son adecuados para modelar las colaboraciones dentro de las páginas. Este nivel de abstracción, extremadamente importante para el diseñador y ejecutor, y debe ser parte del modelo. Podríamos decir que cada página web es una clase UML en la vista de diseño del modelo (vista lógica), y que sus relaciones con otras páginas (asociaciones) representan hipervínculos. La reacción instintiva a este problema

podría ser el estereotipo de cada atributo o la operación en la clase para indicar si es válido en el servidor o cliente.

Para las páginas web, los estereotipos indican que la clase es una abstracción del comportamiento lógico de una página web en el cliente o el servidor. Las dos abstracciones están relacionadas entre sí con una relación direccional entre los dos. Esta asociación es estereotipada como «built», ya que se puede decir que una página de servidor crea una página de cliente como se muestra en la figura 3.17. Cada página web dinámica se construye con una página del servidor. Cada página de cliente es construido por, a lo sumo, una página de servidor único;.

Los hipervínculos se implementan en el sistema como una solicitud para una página web y las páginas web se han diseñado como componentes en la vista de aplicación figura 3.18. Esto es porque un vínculo es realmente una solicitud para una página, no cualquiera de las abstracciones de clase.

Los valores etiquetados se utilizan para definir los parámetros que se pasan junto con una solicitud de vínculo. La Asociación «link» valor "Parámetros" son una lista de nombres de parámetro (y valores opcionales que se espera y utiliza la página de servidor que procesa la solicitud). En la figura 3.18, la página SearchResults contiene un número variable de hipervínculos (0.. *) a la página del servidor de GetProduct, donde cada vínculo tiene un valor diferente para el parámetro productId. La página GetProduct basada en la página ProductDetail del producto especificado por el parámetro productId.

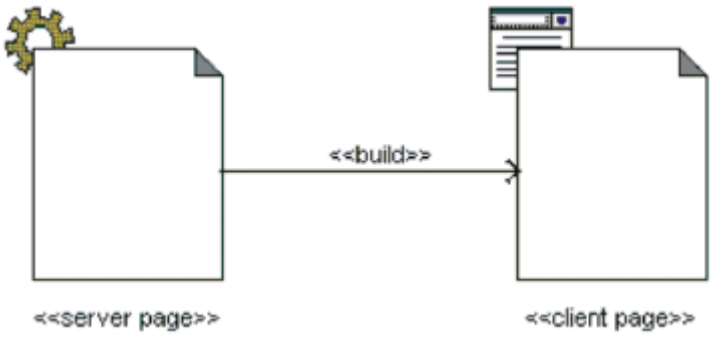


Figura 3.17. Las páginas del servidor crear las páginas del cliente.

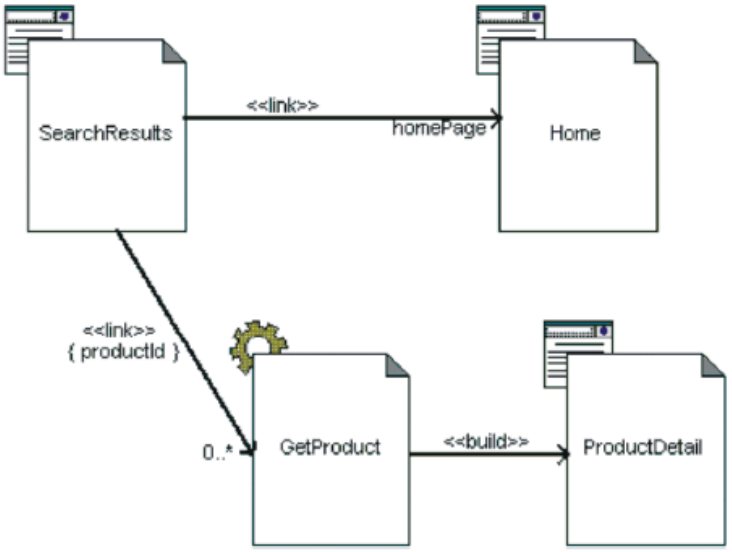


Figura 3.18. La utilización de parámetros hipervínculo.

Las operaciones de la clase «server pages» se convierten en funciones de secuencias de comandos de servidor de la página y sus atributos se convierten en variables para soportar páginas. Las operaciones y atributos de la clase «client pages » asimismo se convierten en funciones y variables visibles en el cliente. Las páginas de cliente se modelan con relación a los recursos del lado cliente: DOM, Applets de Java, controles ActiveX y plug-ins (figura 3.19). Las páginas de servidor se modelan con relación a recursos del lado servidor, componentes de nivel medio, componentes de acceso a bases de datos, sistema operativo etc., ilustran en la figura 3.20.

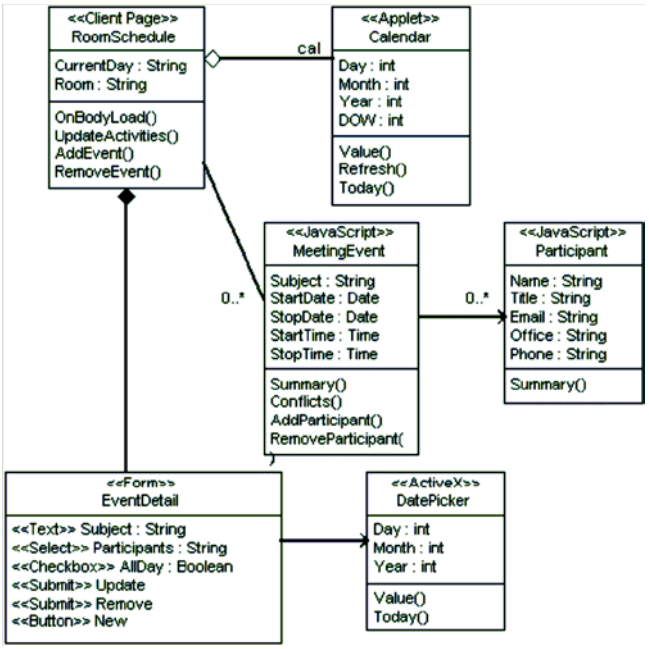


Figura 3.19. Colaboraciones cliente

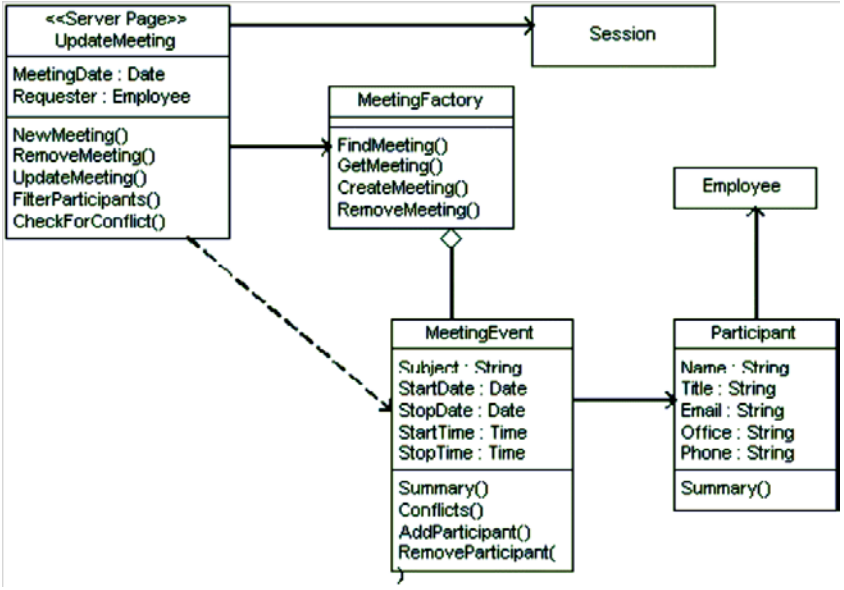


Figure 3.20. Colaboraciones servidor

Una de las mayores ventajas de la utilización de estereotipos de clase para modelar los comportamientos lógicos de páginas web es que sus colaboraciones con los componentes del lado del servidor pueden expresarse de la misma manera como otras colaboraciones de lado de servidor. «server pages» es simplemente otra clase que participa en la lógica de negocio del sistema. En un plano más conceptual, páginas de servidor suelen asumir el papel de los controladores, orquestando la actividad de los objetos del negocio necesarios para alcanzar las metas de negocio iniciadas por solicitud de la página del navegador.

En el lado del cliente, las colaboraciones pueden complicarse un poco. Esto es debido en parte a la variedad de tecnologías que se pueden emplear. Una página de cliente, en su forma más simple, es un documento HTML que contiene la información de presentación. Los Navegadores representan las páginas HTML que siguiendo las instrucciones de formato en la página, a veces con hojas de estilo separadas.

3.2.4 Arquitectura de referencia para sistemas de información

Una arquitectura de sistemas de información consta de los sistemas de información básicos requeridos por las organizaciones para coordinar el comercio mundial y otras actividades. La figura 3.21 ilustra el razonamiento que seguimos e ilustra las principales dimensiones de una arquitectura de sistemas de información internacionales.

FIGURA 3.21. Arquitectura de sistemas de información

Las aplicaciones empresariales suelen tener datos complejos –y la perdida de estos– para trabajar, junto con las reglas del negocio que rompen todas las pruebas de razonamiento lógico. Aunque algunas técnicas y patrones son pertinentes para todo tipo de software, muchos son pertinentes sólo para una rama en particular.

Las aplicaciones empresariales suelen referirse a datos persistentes. Los datos son persistentes porque deben ser más o menos utilizados entre varias ejecuciones de programas. Durante ese tiempo habrá muchos cambios en la estructura de los datos para almacenar piezas de información sin alterar las piezas antiguas. Incluso si hay un cambio fundamental y la empresa instala una aplicación completamente nueva para manejar un proceso, los datos tienen que migrar a la nueva aplicación.

Por lo general hay una gran cantidad de datos hasta el punto de que su gestión es una parte importante del sistema. Los sistemas más antiguos utilizan estructuras de archivos indexados como VSAM IBM y ISAM. Los sistemas modernos suelen utilizar bases de datos relacionales en su mayoría.

Por lo general, muchas personas acceden a los datos simultáneamente. Para muchos sistemas esto puede ser menos de un centenar de personas, pero para sistemas basados en Web esto va por órdenes de magnitud. Incluso sin muchas personas, todavía hay problemas en asegurarse de que dos personas no tienen acceso a los mismos datos al mismo tiempo de manera que provoque errores. Herramientas de gestión de transacciones debe manejar parte de esta carga.

Las aplicaciones empresariales rara vez viven en una isla. Por lo general necesitan integrar con otras aplicaciones empresariales alrededor de la empresa.

Luego está la cuestión de lo que viene bajo el término "lógica del negocio". Esto parece curioso ya que hay pocas cosas que son menos lógicas que la lógica de negocio. Hay que tratar con una matriz fortuita de condiciones extrañas que a menudo interactúan entre sí de manera sorprendente. En esta situación hay que organizar la lógica de negocios tan efectivamente como sea posible, porque lo único cierto es que la lógica va a cambiar con el tiempo.

Gran parte del desafío en el diseño de sistemas de información es en conocer acerca de las alternativas y juzgar las ventajas y desventajas de utilizar una alternativa sobre otra..

La división en capas es una de las técnicas más comunes que los diseñadores de software utilizan para separar un sistema de software complicado.

Las tres capas de Principales

Presentación: Prestación de servicios, visualización de la información (por ejemplo, en Windows o HTML, manejo de solicitud de usuario (clics del mouse, teclado), las solicitudes HTTP, invocaciones de línea de comandos, API)

Dominio: Lógica que es el verdadero punto del sistema

Datos fuente: Comunicación con bases de datos, sistemas, administradores de transacciones, otros paquetes de mensajería

Capas lógicas: dividir un sistema en piezas separadas para reducir el acoplamiento entre las diferentes partes de un sistema. Separación entre capas es útil incluso si las capas están ejecutando en un equipo físico. Sin embargo, hay lugares donde la estructura física de un sistema de marca la diferencia.

En la organización lógica de dominio se puede separar en tres patrones principales:

Un script de transacción: organiza toda esta lógica ante todo como un único procedimiento, por lo que llama directamente a la base de datos a través de una envoltura delgada de base de datos. Cada transacción tendrá su propio script de transacción, a pesar de las subtareas comunes se pueden dividir en subprocedimientos como se muestra en la figura 3.21.

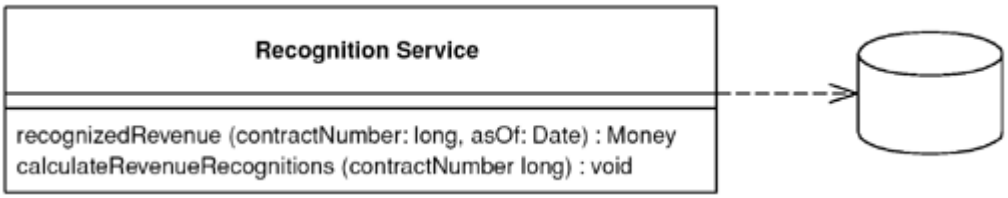


Figura 3.22 Transaction Script; Organiza la lógica empresarial por procedimientos donde cada procedimiento controla una sola solicitud de la presentación

Modelo de dominio La lógica negocio puede ser muy compleja. Reglas y lógica describen muchos casos diferentes y modos de comportamiento ya que esta complejidad es para la cual los objetos fueron diseñados para trabajar. Un modelo de dominio crea una red de objetos interconectados, donde cada objeto representa a algo significativo, ya sea tan grande como una corporación o tan pequeño como una sola línea en un formulario de pedido.

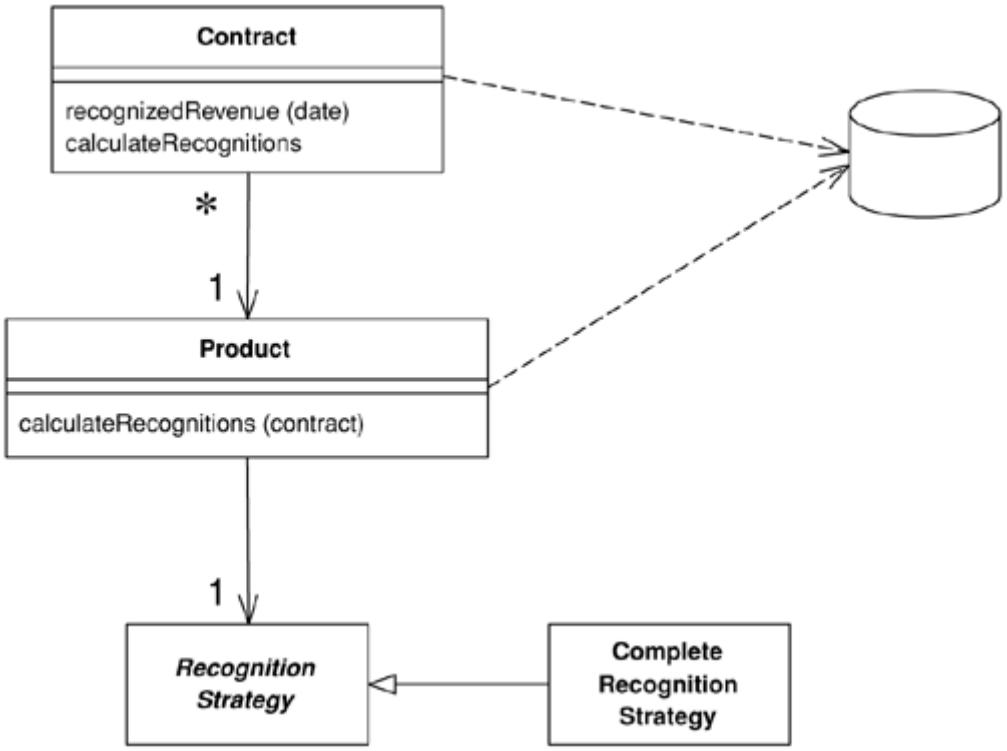


Figura 3.23 Un modelo de objetos del dominio que incorpora datos y comportamiento.

Módulo tabla Uno de los principales mensajes de orientación a objetos es empaquetar los datos con el comportamiento que lo utiliza. El enfoque orientado a objetos tradicional se basa en objetos con identidad, a lo largo de las líneas del modelo de dominio. Así, si tenemos una clase empleado, cualquier instancia de la

misma corresponde a un empleado en particular. Este plan funciona bien porque una vez que tengamos una referencia a un empleado, podemos ejecutar las operaciones, seguir las relaciones y recopilar datos sobre él..

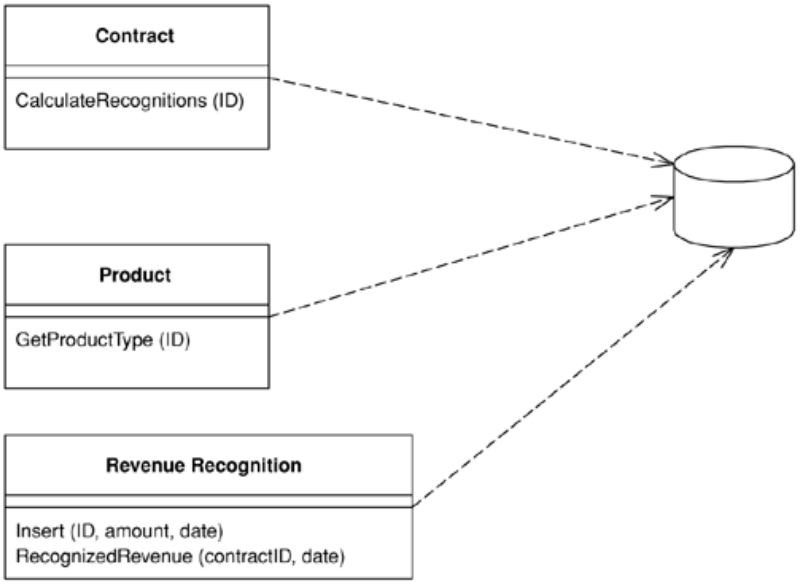


Figura 3.23 Una sola instancia que controla la lógica empresarial de todas las filas de una tabla de base de datos o vista.

Un módulo tabla organiza la lógica de dominio con una clase por cada tabla en la base de datos, y una única instancia de una clase contiene los diversos procedimientos que actuarán en los datos. La principal distinción con el modelo de dominio es que, si tiene muchos pedidos, un modelo de dominio tendrá un objeto de pedido por orden mientras que un módulo de tabla tendrá un objeto para controlar todos los pedidos.

A menudo necesitará comportamiento de varios módulos tabla para hacer un trabajo útil. Muchas veces ves varios módulos de tabla que operan en el mismo registro conjunto (figura 3.24).

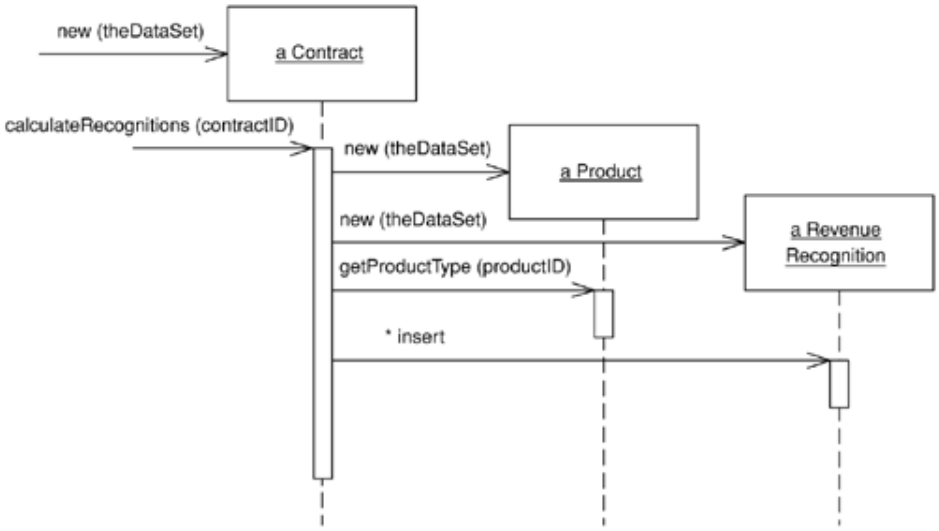


Figura 3.24. Módulos tabla colaborar con un conjunto único de registro

3.2.5 Arquitectura de referencia para ambientes virtuales de aprendizaje

Con la aparición de la Inteligencia Artificial (IA) a principios de los 60 y su aplicación al desarrollo de sistemas de enseñanza se acuño el termino Sistema Inteligente de Tutoría (SIT). Un SIT (Sleeman y Brown, 1982) tiene como objetivo actuar como un tutor humano inteligente, de manera que sea capaz de orientar y enseñar a un estudiante en el proceso de aprendizaje de una materia dada, detectar los errores que el estudiante comete, tratar de determinar el punto en que éste falla para corregirlo y aclarar confusiones o dudas que se le presenten, considerando las peculiaridades de cada estudiante y permitiendo, de esta forma, una enseñanza mas individualizada.

Para poder alcanzar dicho objetivo, durante el desarrollo de un sistema de este tipo deben abordarse cuestiones como la representación en el sistema de la materia objeto de estudio, la representación del conocimiento acerca del estudiante, la estrategia de comunicación con el estudiante, y las estrategias de tutoria a seguir. La arquitectura mas general de un SIT (Wenger, 1987), que puede verse en la Figura 3.25, comprende cuatro componentes principales, cada uno de los cuales viene a dar respuesta a una de las cuestiones anteriores:

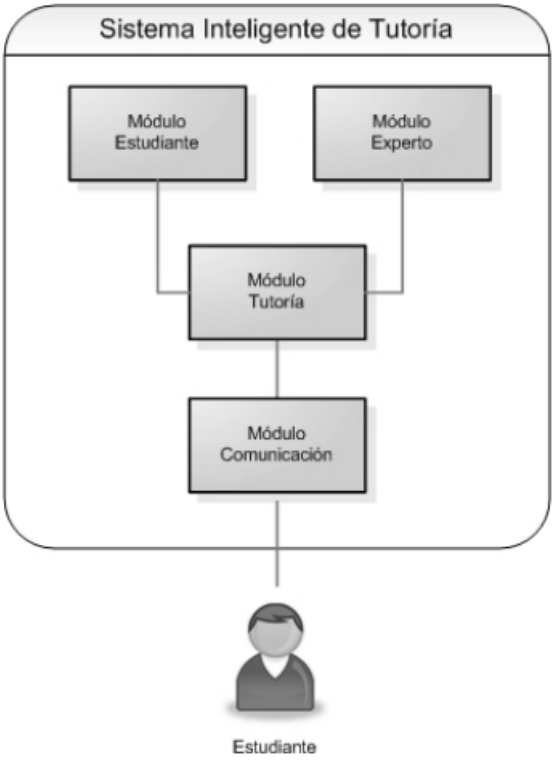


Figura 3.25: Arquitectura clásica de un SIT

Modulo Experto: contiene el conocimiento que el estudiante debeadquirir, y sirve para comprobar el grado de correccion de las accionesy respuestas del estudiante.

Modelo del Estudiante: contiene toda la información que el SIT posee acerca del estudiante: el estado de sus conocimientos actuales, la naturaleza de sus problemas de aprendizaje o su historial desde que empezó a utilizar el sistema. Se emplea para predecir el nivel de comprensión del estudiante y reconocer su forma particular de aprendizaje.

Módulo de Comunicación: soporta la interfaz con el estudiante, proporcionando salidas de la forma más adecuada para cada estudiante e interpretando sus respuestas según las clasificaciones a las que el módulo de tutoría es sensible.

Módulo de Tutoría: contiene las estrategias, reglas y procesos que dirigen las interacciones del sistema con el estudiante, y que permiten al SIT tomar decisiones acerca del material que debe presentar, qué preguntas o ejemplos sugerir y por qué y cuándo se debe interrumpir al estudiante. Se basa en la información proporcionada por el modelo del estudiante.

La manera más clásica de construir un SIT utilizando esta arquitectura se puede asimilar con una típica arquitectura en tres capas, donde el Módulo de Comunicación hace las veces de interfaz de usuario, el Módulo de Tutoría contiene la lógica de la aplicación y los Módulos Experto y del Estudiante serán

sendas bases de datos, una con el conocimiento del experto y otra con la información que se posee sobre el estudiante.

Esta arquitectura, aunque su cliente para un SIT tradicional, no se ajusta a los requisitos de un EV (Entorno virtual) por varios motivos:

Un EV debe servir para poder entrenar a un grupo de estudiantes al mismo tiempo, y no a uno solo, y debe ser capaz de adaptar el entrenamiento a las necesidades de cada alumno, lo cual no está contemplado en la arquitectura de la Figura 3.25.

El Módulo del Estudiante debe modelar el conocimiento de cada estudiante, pero también el del grupo completo.

El estudiante no queda fuera de los límites del sistema, ya que la interacción con el EV se realiza a través de un avatar que se encuentra dentro del propio EV y que está controlado con dispositivos de Realidad Virtual como guantes o sensores. Además, cada estudiante tiene una visión distinta del entorno dependiendo del punto en el que se encuentre.

Aunque será muy intuitivo considerar que el EV forma parte de la interfaz gráfica de usuario, existe una razón que impide hacer esta consideración. Cuando se utiliza una interfaz gráfica clásica, la interacción con la aplicación se realiza a través de menús, botones y controles similares, pero el fin no es aprender a utilizar esos controles, sino que son un medio para comunicarse con la aplicación. Por contra, cuando se utiliza un EV, el fin que se persigue si es aprender a manipular lo que aparece dentro de él. Por este motivo, es necesario que el SIT tenga conocimiento de lo que ocurre dentro del EV, su estado y las posibilidades de interacción

Aunque para el desarrollo de la arquitectura conceptual de un sistema software no es necesario especificar la forma en que se construirá (e.g. utilizando objetos o agentes figura 3.26), en este punto se considera adecuado y necesario introducir los agentes en el diseño arquitectónico. Esto se debe a que sus características particulares pueden obligar a la modificación de la arquitectura si su utilización se deja para más adelante.

Para el diseño de la arquitectura, los distintos autores ofrecen diferentes alternativas, que acaban reduciéndose a dos junto con las distintas variantes que pueden surgir entre ellas.

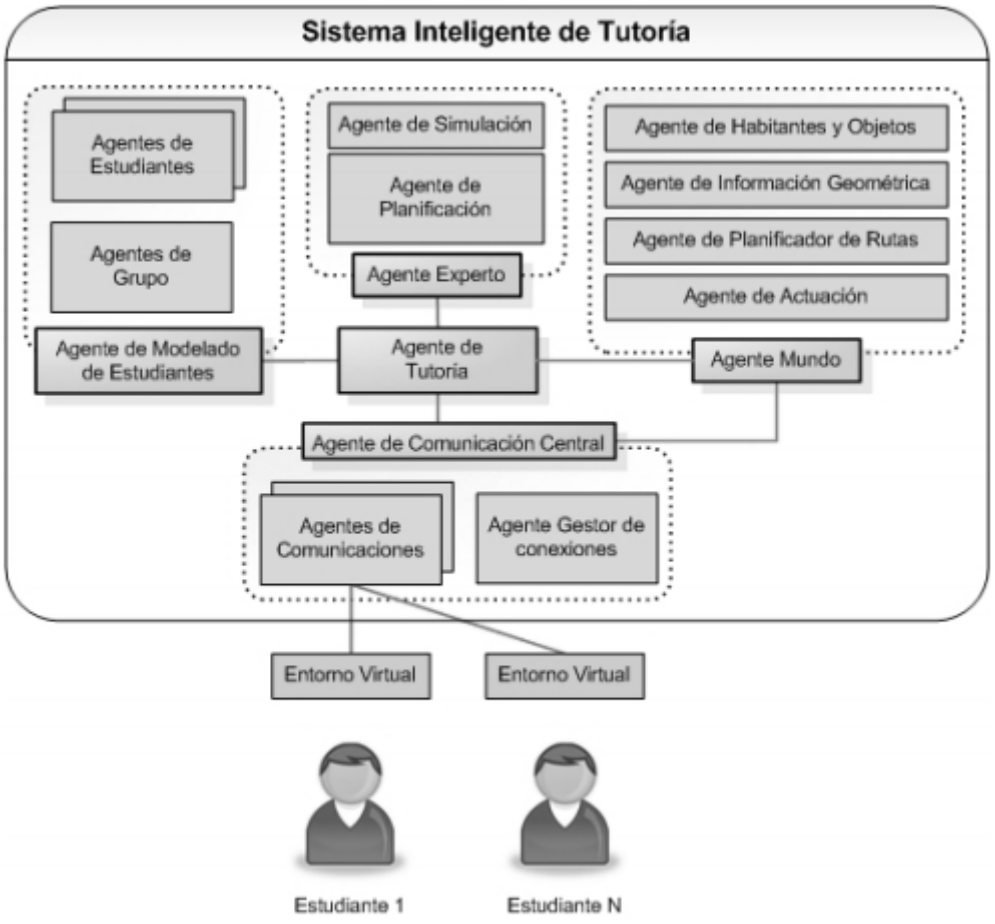


Figura 3.26: Arquitectura basada en agentes

Una de las responsabilidades más complejas que soporta la arquitectura descrita es la relativa a la planificación.

En este sentido, se debe decir que la arquitectura presenta un alto grado de modificabilidad, ya que permite la introducción de nuevos agentes y nuevos operadores para realizar la planificación de manera dinámica, sin modificar el resto del sistema.

Por otra parte, el hecho de realizar la planificación usando una pizarra, todos los agentes se salta en cierta manera la filosofía de utilizar una estructura jerárquica en la que la comunicación entre agentes se realice a través de los agentes supervisores. Uno de los aspectos fundamentales en una arquitectura es que exista la denominada integridad conceptual, que consiste en mantener una homogeneidad de criterios en el diseño de la arquitectura. El hecho de introducir soluciones distintas para cada una de las funciones que debe cumplir la arquitectura redunda en una mayor complejidad de la misma y, a la larga, en una mayor dificultad para entenderla, manejarla y modificarla.

3.2.6 Arquitecturas de referencia para líneas de productos

Qué es una Línea de Productos de Software (LPS)

La idea básica es el ensamblaje de partes de software previamente elaboradas e inspirada en los procesos de producción de sistemas físicos. Producción de aviones, vehículos, computadores, aparatos electrónicos, etc, fundamentada en la reutilización de Software y asumiendo la existencia de una industria de partes

Antecedentes

“La reutilización de software es el proceso de implementar o actualizar sistemas de software usando activos de software existentes” (Sodhi y Sodhi, 1999)

"Reutilización de software es el proceso de crear sistemas de software a partir de software existente, en lugar de desarrollarlo desde el comienzo" (Sametinger, 1997)

Existen varias modalidades de reutilización utilizadas en empresas de software:

- Individual
- Oportunista
- Gestionada:
    - ƒ Institucionalizada, sistemática, planificada, mejorada

Tradicionalmente, la reutilización ha estado basada en oportunidad. Los componentes que se almacenan en un repositorio a la espera de una oportunidad de reutilización.

Las aplicaciones se crean mediante la integración de componentes nuevos, legados o de terceros (COTS) como se muestra en la figura 3.25

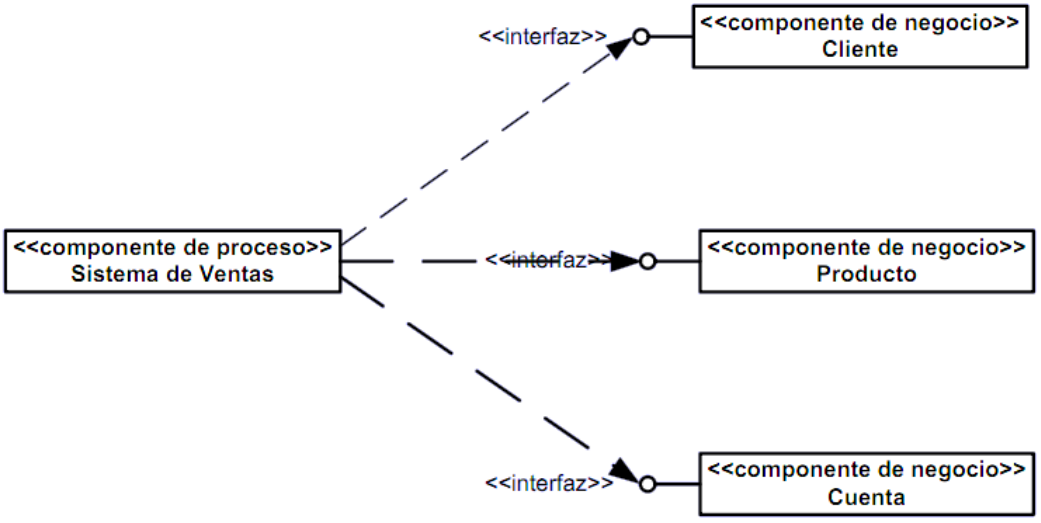


Figura 3.27 modelo de componentes

Definiciones:

Líneas de Productos de Software: se refieren a técnicas de ingeniería para crear un portafolio de sistemas de software similares, a partir de un conjunto compartido de activos de software, usando un medio común de producción" (Krueger, 2006)

Es un conjunto de sistemas de software que comparten un conjunto común y gestionado de aspectos que satisfacen las necesidades específicas de un segmento de mercado o misión y que son desarrollados a partir de un conjunto común de activos fundamentales [de software] de una manera prescrita" (Clements and Northrop, 2002)

"Línea de Productos de Software.consiste de una familia de sistemas de software que tienen una funcionalidad común y alguna funcionalidad variable" (Gomma, 2004) La funcionalidad común descansa en el uso recurrente de un conjunto común de activos reutilizables (requisitos, diseños, componentes, servicios web, etc.) Los activos son reutilizados por todos los miembros de la familia

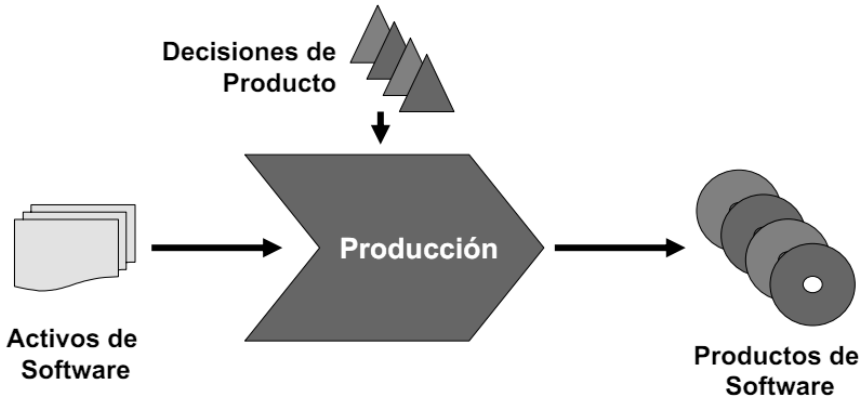


Figura 3.28 Modelo Básico de una Línea de Productos de Software (LPS)

La entrada: Activos de Software son una colección de partes de software (requisitos, diseños, componentes, casos de prueba, etc.) que se configuran y componen de una manera prescrita para producir los productos de la línea

El control: Modelos de Decisión y Decisiones de Productos y los Modelos de Decisiones describen los aspectos variables y opcionales de los productos de la línea cada producto de la línea es definido por un conjunto de decisiones (decisiones del producto)

El proceso de producción establece los mecanismos o pasos para componer y configurar productos a partir de los activos de entrada; las decisiones del producto se usan para determinar que activos de entrada utilizar y como configurar los puntos de variación de esos activos

La salida son el conjunto de todos los productos que pueden o son producidos por la línea de productos en consecuencia la entrega de productos de software de es

más rápida, económica y con una mejor calidad. Las LPS producen mejoras en: tiempo de entrega del producto (time to market), costos de ingeniería, tamaño del portafolio de productos, reducción de las tasas de defectos y calidad de los productos

El paradigma de desarrollo de software LPS requiere que las empresas que lo adopten consideren aspectos conceptuales fundamentales que comprenden la reutilización de software los activos existentes que pueden ser reutilizables los cuales son clasificados por dominios y familias; aspectos tecnológicos fundamentales para desarrollar y mantener activos y productos de software, aspectos metodológicos que proporcione una guía de cómo desarrollar y mantener los activos y productos de software, aspectos organizativos que se refieren a aspectos de organización interna de la empresa organizarse y Aspectos gerenciales que ayudan a gestionar los proyectos de desarrollo de activos y productos.

"Una arquitectura de Líneas de Productos de Software es la estructura o estructuras de un sistema que comprende los componentes del software, las propiedades visibles externamente de estos componentes, y las relaciones entre ellos" (Bass, 1998)

Las propiedades externas de los componentes son sus interfaces (APIs) y z sus características de rendimiento, manipulación de errores, uso compartido de recursos, etc.

La arquitectura de una LPS es una arquitectura de software genérica que describe la estructura de toda la familia de productos y no solamente la de un producto particular y captura los aspectos comunes y variables de una familia de productos de software. Los aspectos comunes de la arquitectura son capturados por los componentes de software que son comunes a toda la familia y los aspectos variables de la arquitectura son capturados por los componentes de software que varían entre los miembros de la familia; también denominada arquitectura de dominio. La arquitectura LPS debe ser instanciada cada vez que se desarrolla un producto de la línea

Una arquitectura LPS es instanciada a través de mecanismos de variabilidad:

Herencia por ejemplo suplantación de un método heredado de una clase en un componente

Puntos de extensión, agregar nueva funcionalidad o comportamiento a un componente

Parametrización, el comportamiento de un componente puede ser parametrizado a tiempo de diseño y definido a tiempo de implementación por ejemplo macros o templates

Configuración: Selección y "des selección" de los componentes de la arquitectura

Selección a tiempo de compilación: La implementación de una funcionalidad es seleccionada, entre varias posibles, al momento de la compilación del componente o de la aplicación

La introducción del paradigma LPS en una empresa de software es un proceso complejo, gradual y lleno de dificultades. Para obtener los beneficios que este paradigma ofrece, una empresa debe tomar en consideración diferentes factores: tecnológicos, metodológicos, organizacionales y gerenciales

4. Pruebas de software

4.1 Definiciones

4.1.1 Prueba, caso de prueba, defecto, falla, error, verificación, validación.

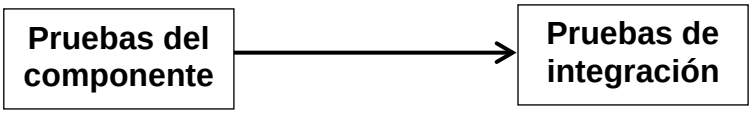
Prueba

La prueba es el proceso de ejecución de un programa con la intención de descubrir un error. (Pressman 2000)

Las pruebas son un conjunto de actividades que se pueden planificar por adelantado y llevar a cabo sistemáticamente. Por esta razón, se debe definir en el proceso de la ingeniería del software una plantilla para las pruebas del software: un conjunto de pasos en los que podamos situar los métodos específicos de diseño de casos de prueba (Somerville 2007 pag. 500).

Un proceso general de pruebas comienza con la prueba de unidades de programas individuales tales como funciones u objetos. A continuación, éstas se integraban en subsistemas y sistemas, y se prueban las interacciones entre estas unidades. Finalmente, después de entregar el sistema, el cliente puede llevar a cabo una serie de pruebas de aceptación para comprobar que el sistema funciona tal y como se ha especificado.

Este modelo de proceso de pruebas es apropiado para el desarrollo de sistemas grandes; pero para sistemas más pequeños, o para sistemas que se desarrollan mediante el uso de scripts o reutilización, a menudo se distinguen menos etapas en el proceso. Una visión más abstracta de las pruebas del software se muestra en la Figura 4.1. Las dos actividades fundamentales de pruebas son la prueba de componentes -probar las partes del sistema- y la prueba del sistema -probar el sistema como un todo.



Desarrollador de software Equipo de pruebas independiente

Figura 4.1 Fase de pruebas

Características generales:

- Las pruebas comienzan a nivel de módulo y trabajan «hacia fuera», hacia la integración de todo el sistema basado en computadora.
- Según el momento, son apropiadas diferentes técnicas de prueba.

- La prueba la lleva a cabo el responsable del desarrollo del software y (para grandes proyectos) un grupo independiente de pruebas.
- La prueba y la depuración son actividades diferentes, pero la depuración se debe incluir en cualquier estrategia de prueba.

Casos de prueba

Conjunto de condiciones o variables bajo las cuáles el analista determinará si el requisito de una aplicación es parcial o completamente satisfactorio.

El objetivo del proceso de diseño de casos de prueba es crear un conjunto de casos de prueba que sean efectivos descubriendo defectos en los programas y muestren que el sistema satisface sus requerimientos.(Somerville: pag:504)

Para diseñar un caso de prueba, se selecciona una característica del sistema o componente que se está probando. A continuación, se selecciona un conjunto de entradas que ejecutan dicha característica, documenta las salidas esperadas o rangos de salida y, donde sea posible, se diseña una prueba automat1zada que prueba que las salidas reales y esperadas son las mismas.

Defecto

Un defecto de software (computer bug en inglés), es el resultado de un fallo o deficiencia durante el proceso de creación de programas de ordenador o computadora (software). Dicho fallo puede presentarse en cualquiera de las etapas del ciclo de vida del software aunque los más evidentes se dan en la etapa de desarrollo y programación. Los errores pueden suceder en cualquier etapa de la creación de software.

Falla

Una falla ocurre cuando un programa no se comporta de manera adecuada. La falla es una propiedad estática de un sistema en ejecución.(Alfredo Weitzenfeld pag. 599)

Error

Es una acción humana que provoca que un software contenga una falta. Un error puede significar la existencia de una falta en el programa lo cual hace que el sistema falle.

Verificación

La verificación se refiere al conjunto de actividades que aseguran que el software implementa correctamente una función específica. La validación se refiere a un conjunto diferente de actividades que aseguran que el software construido se ajusta a los requisitos del cliente. Bohem [BOE81] lo define de otra forma:

- Verificación: «¿Estamos construyendo el producto correctamente
- Validación: «¿Estamos construyendo el producto correcto?»

Validación

La validación del software o, de forma más general, la verificación y validación (V & V) se utiliza para mostrar que el sistema se ajusta a su especificación y que cumple las expectativas del usuario que lo comprará. Implica procesos de comprobación, como las inspecciones y revisiones, en cada etapa del proceso del software desde la definición de requerimientos hasta el desarrollo del programa, Sin embargo, la mayoría de los costos de validación aparecen después de la implementación, cuando se prueba el funcionamiento del sistema,

A excepción de los programas pequeños, los sistemas no se deben probar como una simple unidad monolítica. La Figura 4.2 muestra un proceso de pruebas de tres etapas en el cual se prueban los componentes del sistema, la integración del sistema y, finalmente, el sistema con los datos del cliente. En el mejor de los casos, los defectos se descubren en las etapas iniciales del proceso y los problemas con la interfaz, cuando el sistema se integra. Sin embargo, cuando se descubren defectos el programa debe depurarse y esto puede requerir la repetición de otras etapas del proceso de pruebas. Los errores en los componentes del programa pueden descubrirse durante las pruebas del sistema. Por lo tanto, el proceso es iterativo y se retroalimenta tanto de las últimas etapas como de la primera parte del proceso.

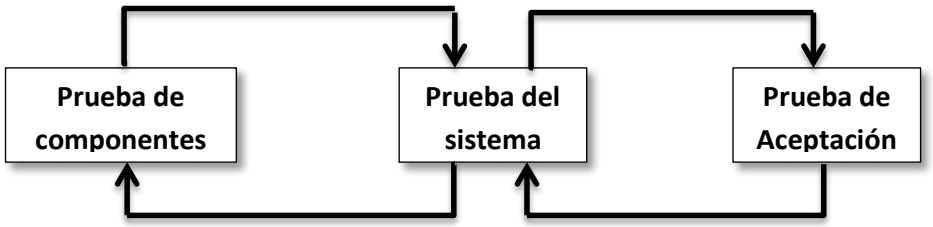


Figura 4.2. Proceso de pruebas

4.1.2 Relación defecto falla error. (Somerville 425)

El software libre de defectos es el software que cumple exactamente con su especificación. Sin embargo, esto no significa que el software nunca falle. Puede haber errores en la especificación que se reflejan en el software. o los usuarios pueden no comprender o no usar bien el sistema software. Sin embargo, eliminar los defectos del software ciertamente tiene un enorme impacto sobre el número de fallos de ejecución del sistema.

Para sistemas de tamaño pequeño y medio, nuestras técnicas de ingeniería del software son tales que probablemente sea posible desarrollar software libre de defectos. Para alcanzar este objetivo, es necesario utilizar varias técnicas de ingeniería del software:

1. **Procesos software confiables.** El uso de un proceso software confiable, con actividades de verificación y validación adecuadas resulta esencial si tiene que minimizarse el número de defectos en el programa, y tienen que detectarse aquellos que se produzcan.

2. **Gestión de calidad.** La organización que desarrolla el sistema debe tener una cultura en la que la calidad guíe el proceso software. Esta cultura debería motivar a los programadores a escribir programas libres de errores. Deberían establecerse estándares de diseño y desarrollo, así como procedimientos para comprobar que dichos estándares se cumplen.

3. **Especificación formal**. Debe realizarse una especificación del sistema precisa (preferiblemente formal) que defina el sistema que se va a implementar. Muchos errores de diseño y programación son el resultado de una mala interpretación de una especificación ambigua o pobremente redactada.

4. **Verificación estática**. Las técnicas de verificación estáticas, como el uso de analizadores estáticos, pueden encontrar características anómalas en el programa que podrían ser defectos. También podría usarse verificación formal, basada en la especificación del sistema.

5. **Tipado fuerte**. Para el desarrollo se debe usar un lenguaje de programación fuertemente tipado, como Java o Ada. Si el lenguaje es fuertemente tipado, el compilador del lenguaje puede detectar muchos errores de programación antes de que puedan ser introducidos en el programa entregado.

6. **Programación segura**. Algunas construcciones de lenguajes de programación son más complejas y propensas a error que otras, y es más probable que se cometan errores si se usan. La programación segura significa evitar, o al menos minimizar, el uso de estas construcciones.

7. **Información protegida**. Debería utilizarse una aproximación para el diseño e implementación del software basada en la ocultación y encapsulamiento de la información. Los lenguajes orientados a objetos como Java, obviamente satisfacen esta condición. Se debería fomentar el desarrollo de programas diseñados para ser legibles y comprensibles.

Existen situaciones en las que es económicamente práctico utilizar todas estas técnicas para crear software libre de defectos. El coste de encontrar y eliminar defectos en el sistema crece exponencialmente a medida que se descubren y eliminan los defectos en el programa como se muestra en la Figura 4.3. A medida que el software se hace más fiable, se necesita emplear más y más tiempo y esfuerzo para encontrar menos y menos defectos. En algún momento, los costes de este esfuerzo adicional se convierten en injustificables.

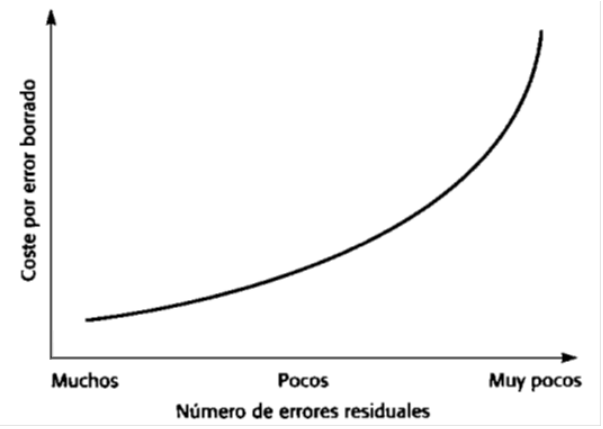


Figura 4.3 relación costo error

4.1.3 Pruebas estructurales, funcionales y aleatorias.

Las pruebas estructurales (Figura 4.4) son una aproximación al diseño de casos de prueba en donde las pruebas se derivan a partir del conocimiento de la estructura e implementación del software. Esta aproximación se denomina a veces pruebas de «caja blanca», de «caja de cristal» o de «caja transparente» para distinguirlas de las pruebas de caja negra.
La comprensión del algoritmo utilizado en un componente puede ayudar a identificar particiones adicionales y casos de prueba.(Somerville 2005 pag: 509)

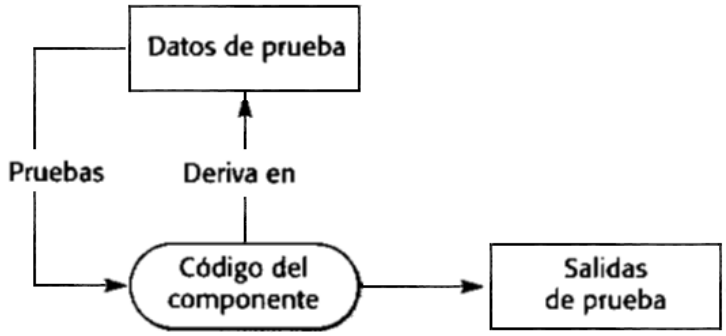


Figura 4.4 pruebas estructurales

Las pruebas funcionales son el proceso de probar una entrega del sistema que será distribuida a los clientes. El principal objetivo de este proceso es incrementar la confianza del suministrador en que el sistema satisface sus requerimientos. Si es así, éste puede entregarse como un producto o ser entregado al cliente. Para demostrar que el sistema satisface sus requerimientos, tiene que mostrarse que éste entrega la funcionalidad especificada, rendimiento y confiabilidad, y que no falla durante su uso normal. Las pruebas de entregas son normalmente un proceso de pruebas de caja negra en las que las pruebas se derivan a partir de la especificación del sistema. El sistema se trata como una caja negra cuyo comportamiento sólo puede ser determinado estudiando sus entradas y sus salidas relacionadas. En este sentido al probador sólo le interesa la funcionalidad

y no la implementación del software. La Figura. 4 ilustra el modelo de un sistema que se admite en las pruebas de caja negra.

El probador presenta las entradas al componente o al sistema y examina las correspondientes salidas. Si las salidas no son las, entonces la prueba ha detectado un problema con el software.

Cuando se prueban las entregas del sistema, debería intentarse «romper» el software eligiendo casos de prueba que pertenecen al conjunto E, en la Figura 23.4. Es decir, el objetivo debería ser seleccionar entradas que tienen una alta probabilidad de generar fallos de ejecución del sistema (salidas del conjunto S,). Se utiliza la experiencia previa de cuáles son las pruebas de defectos que probablemente tendrán éxito y las guías de pruebas ayudarán a elegir la adecuada.

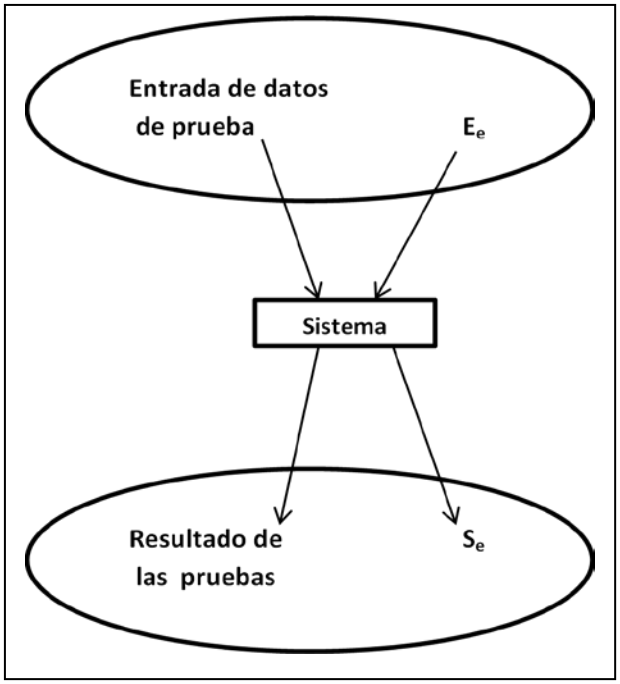


Figura 4.5 pruebas funcionales

La prueba aleatoria genera arbitrariamente los casos de prueba a partir del dominio de entrada de las variables del sistema bajo prueba.

Puede aplicarse a cualquier criterio de suficiencia aunque este no influye en el proceso de generación de casos. Por ello, la generación de casos de prueba es inmediata pero ineficaz. Además está comprobado que el cumplimiento del criterio de suficiencia de los datos generados aleatoriamente es muy dependiente del rango para el cual son generados.

Aunque la prueba aleatoria es aparente mente ineficaz. La generación de casos de prueba aleatorios es eficiente, puesto que solo se necesita un generador de números aleatorios junto con un poco de soporte de software, y permite una automatización inmediata.

La prueba aleatoria como prueba por si sola se ha utilizado en raras ocasiones con objeto de probar software difícil, muy elaborado o muy específico.

La aplicación principal de la prueba aleatoria se aplica en comparaciones con otros métodos de prueba.

4.1.4 Documentación del diseño de las pruebas.

Los esfuerzos realizados en la planificación y preparación de pruebas se resumen en un documento denominado plan de pruebas. En su preparación debe resaltarse la necesidad de que los casos de prueba estén identificados clara y unívocamente, con el fin de facilitar las referencias futuras al ser ejecutados. También es importante detallar concretamente tanto los datos de entrada como las salidas esperadas, que el encargado de su ejecución cotejara posteriormente con la información obtenida directamente del sistema. Tales recomendaciones pueden encontrase en el estándar IEEE 829; 1998. No obstante cada organización debe definis los documentos que quiere producir en sus pruebas con el objeto que la productividad de estos procesos sea tan alta como sea posible. Notese que el plan de pruebas no es un fin en sí mismo, sino un medio para desarrollar ordenadamente los procesos de prueba.

4.2 Proceso de pruebas.

4.2.1 Generar un plan de pruebas

Objetivo del documento

Señalar el enfoque, los recursos y el esquema de actividades de prueba, así como los elementos a probar, las características, las actividades de prueba, el personal responsable y los riesgos asociados

ESTRUCTURA FIJADA EN EL ESTANDAR

1. Identificador único del documento
2. Introducción y resumen de elementos y características a probar
3. Elementos software a probar
4. Características a probar
5. Características que no se probarán
6. Enfoque general de la prueba
7. Criterios de paso/fallo para cada elemento
8. Criterios de suspensión y requisitos de reanudación

9. Documentos a entregar
10. Actividades de preparación y ejecución de pruebas
11. Necesidades de entorno
12. Responsabilidades en la organización y realización de las pruebas
13. Necesidades de personal y formación
14. Esquema de tiempos
15. Riesgos asumidos por el plan y planes de contingencias
16. Aprobaciones y firmas con nombre y puesto desempeñado

4.2.2 Diseñar pruebas específicas.

Objetivo del documento

Especificar los refinamientos necesarios sobre el enfoque general reflejado en el plan e identificar las características que se deben probar con este diseño de pruebas

ESTRUCTURA FIJADA EN EL ESTANDAR

1. Identificador único para la especificación. Proporcionar también una referencia del plan asociado (si existe)
2. Características a probar de los elementos software (y combinaciones de características)
3. Detalles sobre el plan de pruebas del que surge este diseño, incluyendo las técnicas de prueba específica y los métodos de análisis de resultados
4. Identificación de cada prueba:
    a. identificador
    b. casos que se van a utilizar
    c. procedimientos que se van a seguir
5. Criterios de paso/fallo de la prueba (criterios para determinar si una característica o combinación de características ha pasado con éxito la prueba o no)

4.2.3 Tomar configuración del software a probar.

Establecer la configuración de la instalación a llevar acabo, determinando que componentes deben instalarse en cada equipo de hardware, y como debe realizarse dicha instalación. En muchas ocasiones es recomendable realizarse un manual de instalación y configuración, donde se recojan las operaciones de instalación del software y su posterior configuración con los parámetros adecuados de o las maquinasy software base donde se procederá a instalar.

4.2.4 Configurar las pruebas

Un elemento importante del proceso de validación es la revisión de la configuración. La intención de la revisión es asegurarse de que todos los elementos de la configuración del software se han desarrollado apropiadamente, se han catalogado y están suficientemente detallados para soportar la fase de mantenimiento durante el ciclo de vida del software. La revisión de la configuración, a veces denominada auditoría,

4.2.5 Evaluar resultados.

4.2.5.1 Depuración.

La depuración ocurre como consecuencia de una prueba efectiva. Es decir, cuando un caso de prueba descubre un error, la depuración es el proceso que provoca la eliminación del error. Aunque la depuración puede y debe ser un proceso ordenado, sigue teniendo mucho de arte. Un ingeniero del software, al evaluar los resultados de una prueba, se encuentra frecuentemente con una indicación «sintomática» de un problema en el software.

Como se muestra en la Figura 4.7, el proceso de depuración comienza con la ejecución de un caso de prueba. Se evalúan los resultados y aparece una falta de correspondencia entre los esperados y los encontrados realmente. En muchos casos, los datos que no concuerdan son un síntoma de una causa subyacente que todavía permanece oculta. El proceso de depuración intenta hacer corresponder el sistema con una causa, llevando así a la corrección del error.

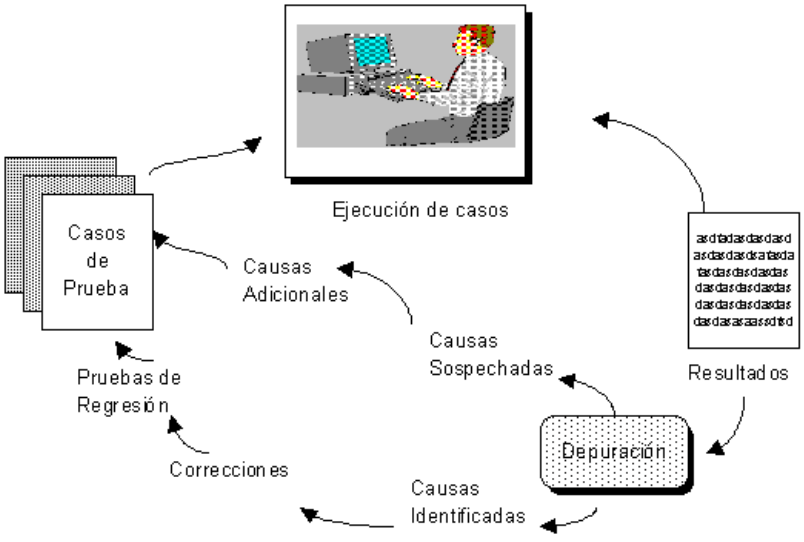


Figura 4.7 proceso de depuración de errores de software

4.2.5.2 Análisis de errores

El código de software escrito deficientemente ocaciona mucho trabajo adicional para el personal de desarrollo de sistemas, lo mantiene ocupado en tareas de mantenimiento y agrava la escasez de recursos para el desarrollo de sistemas

Los errores de software pueden causar perdidadde ingresos y oportunidades de mercado así como procesamiento incorrectode la información –y todo ello reduce las utilidades de la una empresa–,

Los tipos de fallos que pueden ocurrir son específicos del sistema y las consecuencias de un fallo del sistema dependen de la naturaleza de ese fallo. Cuando se redacte una especificación de fiabilidad, habría que identificar los diferentes tipos de fallos, y pensar sobre si éstos deberían ser tratados de fonna diferente en la especificación. En la Figura 9.11 se muestran ejemplos de diferentes tipos de fallos. Obviamente pueden ocurrir combinaciones de éstos, como por ejemplo un fallo que sea transitorio, recuperable y corruptivo.

La mayoría de los sistemas grandes están compuestos por varios subsistemas con diferentes requerimientos de fiabilidad. Puesto que el software que tiene una fiabilidad alta es caro, debería realizarse una valoración de los requerimientos de fiabilidad para cada subsistema por separado en lugar de imponer el mismo requerimiento de fiabilidad para todos los subsistemas. Esto evita imponer altos requerimientos de fiabilidad en aquellos subsistemas en los que no es necesario.

Los pasos que se requieren para establecer una especificación de la fiabilidad son:

1. Para cada subsistema. identificar los tipos de fallos de funcionamiento del sistema que pueden ocurrir y analizar las consecuencias de dichos fallos.
2. A partir del análisis de fallos del sistema. dividir los fallos en clases. Un punto de partida razonable es utilizar los tipos de fallos mostrados en la Figura 4.8.

Clase de fallo	Descripción
Transitorio	Ocurre solamente con ciertas entradas.
Permanente	Ocurre con todas las entradas.
Recuperable	El sistema puede recuperarse sin la intervención del operador.
Irrecuperable	Es necesaria la intervención del operador para recuperarse del fallo.
No corruptivo	El fallo no corrompe el estado del sistema o los datos.
Corruptivo	El fallo corrompe el estado del sistema o los datos.

Figura 4.8Clasificación de fallos de funcionamiento

3. Para cada clase de fallo identificada, definir el requerimiento de fiabilidad utilizando una métrica de fiabilidad adecuada. No es necesario utilizar la misma métrica para diferentes clases de fallos. Si un fallo requiere alguna intervención para poder recuperar el sistema, la métrica más apropiada podría ser la probabilidad de fallos en la petición. Cuando es posible la recuperación automática y el efecto del fallo es la causa de una molestia para el usuario, ROCOF podría ser la más adecuada.
4. Donde sea apropiado, identificar los requerimientos de fiabilidad funcionales que definen la funcionalidad del sistema para reducir la probabilidad de fallos críticos.

4.3 Técnicas de diseño de casos de prueba

El diseño de pruebas para el software o para otros productos de ingeniería puede requerir tanto esfuerzo como el propio diseño inicial del producto. Sin embargo, los ingenieros del software, a menudo tratan las pruebas como algo sin importancia, desarrollando casos de prueba que «parezcan adecuados», pero que tienen poca garantía de ser completos. Recordando el objetivo de las pruebas, debemos diseñar pruebas que tengan la mayor probabilidad de encontrar el mayor número de errores con la mínima cantidad de esfuerzo y tiempo posible.

Cualquier producto de ingeniería puede probarse de una de estas dos formas: (1) conociendo la función específica para la que fue diseñado el producto, se pueden llevar a cabo pruebas que demuestren que cada función es completamente operativa y, al mismo, tiempo buscando errores en cada función; (2) conociendo el funcionamiento del producto, se pueden desarrollar pruebas que aseguren que «todas las piezas encajan», o sea, que la operación interna se ajusta a las especificaciones y que todos los componentes internos se han comprobado de forma adecuada. El primer enfoque de prueba se denomina prueba de caja negra y el segundo, prueba de caja blanca.

Cuando se considera el software de computadora, la prueba de caja negra se refiere a las pruebas que se llevan a cabo sobre la interfaz del software. O sea, los casos de prueba pretenden demostrar que las funciones del software son operativas, que la entrada se acepta de forma adecuada y que se produce un resultado correcto, así como que la integridad de la información externa (por ejemplo, archivos de datos) se mantiene. Una prueba de caja negra examina algunos aspectos del modelo fundamental del sistema sin tener mucho en cuenta la estructura lógica interna del software.

La prueba de caja blanca del software se basa en el minucioso examen de los detalles procedimentales. Se comprueban los caminos lógicos del software proponiendo casos de prueba que ejerciten conjuntos específicos de condiciones y/o bucles. Se puede examinar el «estado del programa» en varios puntos para determinar si el estado real coincide con el esperado o mencionado.

A primera vista parecería que una prueba de caja blanca muy profunda nos llevaría a tener «programas cien por cien correctos». Todo lo que tenemos que hacer es definir todos los caminos lógicos, desarrollar casos de prueba que los ejerciten y evaluar los resultados, es decir, generar casos de prueba que ejerciten exhaustivamente la lógica del programa.

Desgraciadamente, la prueba exhaustiva presenta ciertos problemas logísticos. Incluso para pequeños programas, el número de caminos lógicos posibles puede ser enorme.

Se pueden elegir y ejercitar una serie de caminos lógicos importantes. Se pueden comprobar las estructuras de datos más importantes para verificar su validez. Se pueden combinar los atributos de la prueba de caja blanca así como los de caja negra, para llegar a un método que valide la interfaz del software y asegure selectivamente que el funcionamiento interno del software es correcto

4.4 Enfoque práctico recomendado para el diseño de casos.

Un enfoque práctico debe especificar los pasos para la ejecución de un conjunto de casos de prueba o, más generalmente, los pasos utilizados para analizar un elemento software con el propósito de evaluar un conjunto de características del mismo

Estructura fijada en el estandar

1. Identificador único de la especificación y referencia a la correspondiente especificación de diseño de prueba
2. Objetivo del procedimiento y lista de casos que se ejecutan con él
3. Requisitos especiales para la ejecución (por ejemplo, entorno especial o personal especial)
4. Pasos en el procedimiento. Además de la manera de registrar los resultados y los incidentes de la ejecución, se debe especificar:
   a) La secuencia necesaria de acciones para preparar la ejecución
   b) Acciones necesarias para empezar la ejecución
   c) Acciones necesarias durante la ejecución
   d) Cómo se realizarán las medidas (por ejemplo, el tiempo de respuesta)
   e) Acciones necesarias para suspender la prueba (cuando los acontecimientos no previstos lo obliguen)
   f) Puntos para reinicio de la ejecución y acciones necesarias para el reinicio en estos puntos
   g) Acciones necesarias para detener ordenadamente la ejecución
   h) Acciones necesarias para restaurar el entorno y dejarlo en la situación existente antes de las pruebas
   i) Acciones necesarias para tratar los acontecimientos anómalos

4.5 Estrategias de aplicación de las pruebas.

4.5.1 Pruebas unitarias

La prueba de unidad centra el proceso de verificación en la menor unidad del diseño del software: el componente software o módulo.

Las pruebas que se dan como parte de la prueba de unidad están esquemáticamente ilustradas en la Figura 4.9. Se prueba la interfaz del módulo para asegurar que la información fluye de forma adecuada hacia y desde la unidad de programa que está siendo probada. Se examinan las estructuras de datos locales para asegurar que los datos que se mantienen temporalmente conservan su integridad durante todos los pasos de ejecución del algoritmo. Se prueban las condiciones límite para asegurar que el módulo funciona correctamente en los límites establecidos como restricciones de procesamiento.

Se ejercitan todos los caminos independientes (caminos básicos) de la estructura de control con el fin de asegurar que todas las sentencias del módulo se ejecutan por lo menos una vez. Y, finalmente, se prueban todos los caminos de manejo de errores.

Antes de iniciar cualquier otra prueba es preciso probar el flujo de datos de la interfaz del módulo. Si los datos no entran correctamente, todas las demás pruebas no tienen sentido. Además de las estructuras de datos locales, durante la prueba de unidad se debe comprobar (en la medida de lo posible) el impacto de los datos globales sobre el módulo.

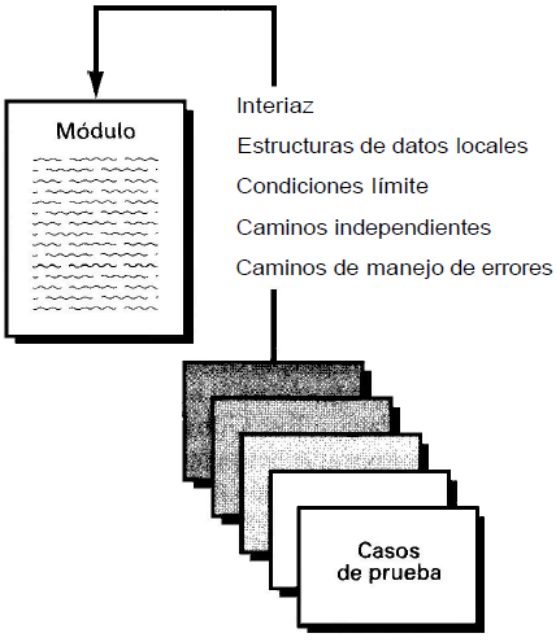


FIGURA 4.9 Prueba de unidad

Durante la prueba de unidad, la comprobación selectiva de los caminos de ejecución es una tarea esencial. Se deben diseñar casos de prueba para detectar errores debidos a cálculos incorrectos, comparaciones incorrectas o flujos de control inapropiados. Las pruebas del camino básico y de bucles son técnicas muy efectivas para descubrir una gran cantidad de errores en los caminos.

4.5.2 Prueba de integración

La prueba de integración se dirige a todos los aspectos asociados con el doble problema de verificación y de construcción del programa. Durante la integración, las técnicas que más prevalecen son las de diseño de casos de prueba de caja negra, aunque se pueden llevar a cabo algunas pruebas de caja blanca con el fin de asegurar que se cubren los principales caminos de control.

Las pruebas de integración parten de los componentes individuales previamente probados y tienen como objetivo descubrir errores que se, pueden producir en la interacción entre los módulos. En teoría; la combinación de componente válidos debería dar como resultado software en el que no se delectan errores, pero en la práctica hay múltiples ocasiones en las que las pruebas de unidad no detectan errores que si se revelan al ejecutar pruebas sobre el software integrado.

Las interfaces entre los módulos suelen ser fuente de errores de integración, pero también los tratamientos concurrentes, como el acceso a estructuras de datos comunes o la definición de transacciones sobre las bases de datos. En otros

En la integración incremental. El programa se construye y se prueba en pequeños segmentos en los que los errores son más fáciles de aislar y de corregir, es más probable que se puedan probar completamente las interfaces y se puede aplicar un enfoque de prueba sistemática.

La prueba de integración descendente es un planteamiento incremental a la construcción de la estructura de programas. Se integran los módulos moviéndose hacia abajo por la jerarquía de control, comenzando por el módulo de control principal (programa principal). Los módulos subordinados (subordinados de cualquier modo) al módulo de control principal se van incorporando en la estructura, bien de forma primero-en-profundidad, o bien de forma primero-en-anchura.

Como se muestra en la Figura 4.10, la integración primero-en-profundidad integra todos los módulos de un camino de control principal de la estructura. La selección del camino principal es, de alguna manera, arbitraria y dependerá de las características específicas de la aplicación.

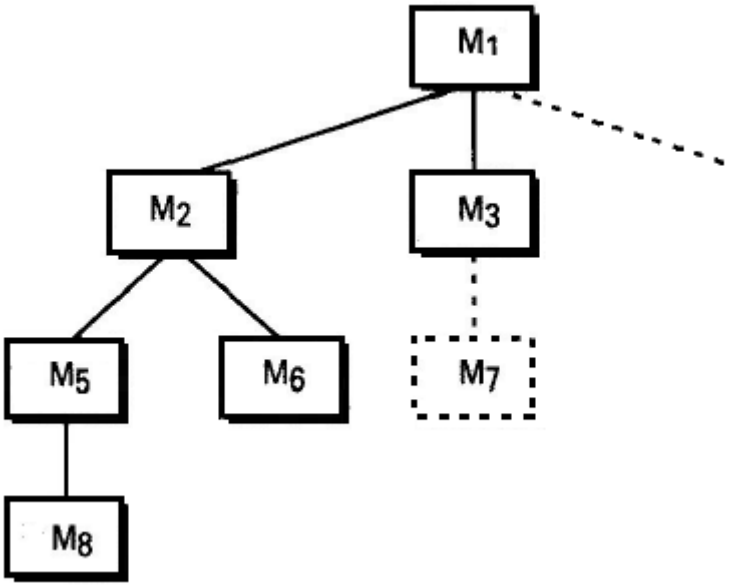


FIGURA 4.10. Integración descendente

La estrategia de integración descendente verifica los puntos de decisión o de control principales al principio del proceso de prueba. En una estructura de programa bien fabricada, la toma de decisiones se da en los niveles superiores de la jerarquía y, por tanto, se encuentran antes. Si existen problemas generales de control, es esencial reconocerlos cuanto antes. Si se selecciona la integración primero en profundidad, se puede ir implementando y demostrando las funciones completas del software.

4.5.3 Del sistema.

Los objetivos de las pruebas de sistema son la detección de errores en el producto ya integrado. En ellas se ejercitan condiciones que anteriormente habría sido difícil comprobar, como las interfaces entre subsistemas, la comunicación con el entorno o el cumplimiento de los requisitos no funcionales. En la Tabla 3.3 se resumen los objetivos de los tipos de pruebas de sistema más importantes.

Cabe destacar que, aunque el producto haya sido realizado a medida para ser implantado en una instalación concreta, las pruebas de sistema se ejecutan también en entornos de prueba controlados.

Tipo de pruebas	Breve descripción de sus objetivos
Funcionales	Orientadas a detectar si el software implementa adecuadamente la funcionalidad descrita en los requisitos.
De comunicaciones	Se prueban las interfaces de comunicación entre diferentes componentes del sistema.
Rendimiento	Se determina si los tiempos de respuesta del sistema, tanto en condiciones normales como en condiciones especiales, se encuentran dentro de los límites predefinidos.
Volumen	Dedicadas a probar el software trabajando con grandes cantidades de datos, similares a las esperadas en producción.
Sobrecarga	Se prueba el sistema con cargas masivas de trabajo. Permiten identificar los límites soportables por el software y eliminar comportamientos indeseados en situaciones de sobrecarga.
Disponibilidad	Consisten en probar la disponibilidad del sistema ante fallos de diferentes componentes de la arquitectura, físicos o lógicos.
Utilizabilidad o usabilidad	Se intenta determinar si el sistema será fácil de utilizar para sus usuarios.
De entorno	Orientadas a probar las interacciones entre el sistema y otros sistemas de su entorno.
Seguridad *(security)*	Se tratará de violar toda protección del sistema, por ejemplo probando aspectos relativos a la confidencialidad o integridad de la información.
Seguridad *(safety)*	Se prueba si el sistema puede generar riesgos de seguridad en su entorno físico.

Figura 4.11 pruebas de sistema

4.5.4 Pruebas de aceptación

Estas pruebas se realizan para que el cliente certifique que el sistema es válido para él. La planificación detallada de estas pruebas debe haberse realizado en etapas tempranas del desarrollo del proyecto, con el objetivo de utilizar los resultados como indicador de su validez: si se ejecutan las pruebas documentadas a satisfacción del cliente, el producto se considera correcto y. por tanto, adecuado para su puesta en producción.

5. Implantación y mantenimiento.

5.1 Implantación e Integración de casos de uso y componentes de software.

El modelo del análisis proporciona una comprensión detallada de los requisitos e impone una estructura del sistema a mantener cuando se le deforma. El diseño

produce un modelo físico del sistema a implementar (modelo del diseño) que es no genérico (específico para una implementación), más formal que el del análisis, dinámico, realizado según la ingeniería de ida i vuelta con el modelo de implementación y que debe ser mantenido durante todo el ciclo de vida del software.

El proceso unificado plantea las siguientes actividades del diseño:

a) Diseñar la arquitectura: consiste en esbozar los modelos de diseño y de despliegue (distribución física de sistema entre los nodos-procesadores-de computo) y en diseñar su arquitectura (vistas arquitectónicas delos modelos de diseño y de despliegue).
b) Diseñar los casos de uso: Consiste en identificar las clases de diseño y los subsistemas, didtribuir el comportamiento del caso de uso entre las clases y subsistemas participantes, definir los requisitos sobre las operaciones de las clases de diseño y los subsistemas, y capturar los requisitos especiales (no funcionales) del caso de uso.
c) Diseñar las clases: Consiste en definir para cada clase de diseño; sus operaciones, atributos, relaciones en las que participa, sus métodos, los estados de los objetos de la clase (definidos mediante diagramas de estados) y los requisitos especiales relevantes a su implementacion.
d) Diseñar los subsistemas: Consiste en definir las dependencias que se producen entre subsistemas cuando una clase pertenece a varios, las interfaces proporcionadas por los subsistemas (cada interfaz debe estar soportada por una interfaz de la clase de diseño), y los contenidos de los subsistemas (las clases que contiene y las operaciones que realiza).

Implementación consiste en implementar el sistema en términos de componentes, es decir, ficheros de código fuente, de cabecera ejecutables etc. Los fines de la implementación son básicamente; planificar las integraciones a realizar en cada iteración (produciendo un sistema que se inplementa en pequeños pasos), distribuir los componentes ejecutables a los nodos en el diagrama de despliegue, implementar las clases y subsistemas encontrados durante el diseño por medio de componentes de ficheros fuentes, probar los componenetes individualmente e integrarlos en el sistema.

El proceso unificado plantea las siguientes actividades de implementación:

a) Implementar la arquitectura: Consiste en esbozar el modelo de implementación y sus arquitectura; identificando los componentes principales (Ejemplo. Los componentes ejecutables) y asignando estos componentes a los nodos de la red.
b) Integrar el sistema: Consiste en planificar cada construcción (de cada caso de uso) y y en integrar cada construcción en el sistema.
c) Implementar los subsistemas: Consiste en comprobar que los requisitos de los subsistemas son implementados correctamente por los componentes.

d) Implementar las clases: Consiste en implementar cada clase de diseño en un componente fichero. Esto incluye ; definir el componente fichero con sus clases, generar el código fuente de los prototipos de las clases, implementar las operaciones por medio de métodos, comprobar que componente tiene las mismas interfaces que las clase de diseño que contiene.
e) Realizar pruebas de unidad: consiste en probar cada componente implementado como unidad individual, realizando pruebas de especificación (pruebas de caja negra) y pruebas de estructura (pruebas de caja blanca).

5.2 Mantenimiento del software.

El mantenimiento es la fase más costosa, más larga y más importante dentro de la vida de una aplicación informática. Su importancia está en su larga duración que es la garantía de que la aplicación funciona correctamente. Por ser la última fase del ciclo de vida paga las consecuencias de una mala programación y paga también los errores cometidos en las fases anteriores.

Los problemas más Importantes son:

- Escasa implicación del usuario en el desarrollo de la aplicación, lo que implica una mayor necesidades posteriores cambios en la aplicación.
- Documentación incompleta de las fases anteriores lo que implica mayor dificultad para realizar y evaluar los cambios en la aplicación.
- La fase de mantenimiento esta mal considerada. se considera menos creativa y más sencilla, se le dedica tiempo insuficiente, pocos recursos y poco esfuerzo.

Factores que inciden en la complejidad del mantenimiento:

- No existen métodos ni técnicas concretas para realizar las operaciones asociadas a esta rase.
- La informática evoluciona muy rápidamente produciendo un rápido desfase de los productos.

Funciones del mantenimiento

El objetivo del mantenimiento es asegurar el rendimiento de una aplicación.

Las funciones son:

Corregir defectos: Esta función se realiza en el momento que se detectan los defectos. Es la fusión mas extendida y algunos autores consideran mantenimiento solo esta parte.

Mejorar el rendimiento: Es decir, conseguir que las aplicaciones que funcionan correctamente lo haga lo mejor posible.

Adaptar el sistema o la aplicación a un entorno cambiante.

Tipos de mantenimiento

Estos son los diferentes tipos de mantenimiento:

- Correctivo: Soluciona los problemas y se detectan a lo largo de la vida de la aplicación.
- Evolutivo: Cubre las nuevas necesidades del usuario que no fueron detecladas en la fase de análisis del ciclo de vida.
- Adaptativo: Adapta la aplicación a cambios en el entorno tecnológico del trabajo, generalmente es el cambio de Sistema Operativo.
- Preventivo: Actúa sobre las aplicaciones para mejorar su calidad interna o sobre los sistemas para prevenir problemas físicos que puedan crear fallos o deterioro de la aplicación.

Mantenimiento Correctivo

Consiste en resolución de los problemas en la vida de la aplicación. En ocasiones también se incluye la solución de Hardware pero eso depende del contrato entre el cliente y el proveedor. Se realiza a iniciativa de los usuarios que son los que detectan los problemas. Los problemas que se resuelven con este mantenimiento pueden ser consecuencia de algún defecto de desarrollo, de improvisión o de necesidades no detectadas en su momento.

Estos problemas se agrupan en tres clases:

- De tratamiento: Son aquellos en los que la aplicación no funciona correclamente.
- De rendimiento: La aplicación funciona pero el rendimiento no es el deseado.
- De diseño: La aplicación funciona pero hay cosas que no hace como quiere el usuario.

Mantenimiento Evolutivo

Consiste en mejorar la aplicación para que desarrolle nuevas tareas no previstas en un principio. El cliente y el equipo de mantenimiento deberán estar de acuerdo en el hecho de que no se han producido errores, ni por parte del cliente ni del desarrollador, sino que estos nuevos requisitos de la aplicación se producen por una nueva necesidad del cliente.

Esta función de mantenimiento se hace a petición del usuario. Puede ser posible que el equipo de mantenimiento sugiera cambios en la aplicación para adaptar el programa a una situación nueva o para mejorar su funcionamiento.

Los problemas relacionados con el mantenimiento evolutivo son todos aquellos derivados de cambios en el entorno de trabajo. Los cambios más frecuentes son:

La empresa cambia la forma de trabajar.

Algún proveedor del cliente fuerza algún tipo de cambio en la aplicación. Se producen cambios en el entorno de la empresa (legislativos. etc.), las acciones que se realizan son correctivas. El programa debe modificarse para solucionar los problemas. Si no es posible modificarlo hay que sustituir la aplicación.

Mantenimiento Adaptativo

Consiste en la adaptación de la aplicación a cambios tecnológicos en el entorno de trabajo sobre el que se implanta en su momento. Este tipo de mantenimiento puede realizarse a iniciativa del usuario pero normalmente se produce a petición del equipo de mantenimiento.

El mantenimiento Adaptativo no realiza estrictamente la función de resolver problemas, lo que hace es adaptar y proporcionar mejoras a la situación actual.

Los cambios pueden ser de tipo Hardware o Software.

Las acciones que se deben realizar consisten en la actualización de la aplicación a la versión adaptada al nuevo entorno.

Mantenimiento Preventivo

Dentro del mantenimiento preventivo caben todas las intervenciones realizadas sobre la aplicación y en ocasiones también sobre el entorno de trabajo antes de que se detecte ningún problema y con el objetivo de evitar su aparición.

El mantenimiento preventivo sobre la aplicación suele ser iniciativa de la empresa desarrolladora y el mantenimiento sobre el sistema informático se negocia entre el cliente y el equipo de mantenimiento.

Lo que busca este tipo de mantenimiento es la prevención, para conocer los tipos de problema que se intentan prevenir. El mantenimiento preventivo puede incluir desde actualizaciones de diversos componentes hasta cuestiones de seguridad de usuarios y datos. El mantenimiento preventivo sobre la aplicación consiste en proporcionar nuevas versiones con capacidades mejoradas y el mantenimiento sobre el sistema informático incluye acciones variadas como pueden ser actualizaciones del Hardware, revisiones del SO, optimización del disco duro, funciones de seguridad, etc.

Asistencia al usuario

Existen varias tareas que pueden considerarse parte del mantenimiento pero no pueden incluirse en ninguno de los tipos anteriores. Son las tareas de asistencia al usuario en su trabajo diario. Existen dos tipos:

- Ayuda On-Line.
- Soporte Técnico.

Ayuda On-line, consiste en los diferentes mecanismos para ayudar al usuario mientras esta trabajando en la aplicación. Algunos de estos mecanismos son internos a la aplicación y otros son externos. Los más importantes son:

- Visitas Guiadas: Es un tipo de ayuda que tiene la propia aplicación que nos permite visualizar mediante diapositivas o animaciones las operaciones más importantes que puede realizar la aplicación
- Ayudas de contexto: Ayuda que aparece al colocar el cursor sobre un elemento de la pantalla. es lo que se denomina ayuda sensible al contexlo. También aparece esta ayuda pulsando el botón derecho del ratón sobre el elemento deseado.
- Manuales electrónicos: son esos manuales de estilo clásico que contiene la aplicación con un formato consultable desde dicha aplicación.
- FAQ: lista de consultas que los usuarios realizan frecuentemente, junto a estas preguntas se incluyen las respuestas, soluciones. Son muy típicas en ciertos tipos de páginas Web.
- Glosarios o buscadores de palabras: Ofrecen información relacionada con una o varias palabras ctave. Este tipo de ayuda ofrece dos opciones. La primera. el usuario ofrece la palabra clave y el buscador intentará localizar la ayuda para esa palabra. La segunda es que el buscador ofrece una lista de términos colocadas alfabéticamente y el usuario elige la que desea buscar.
- Manuales en papel: Tienden a desaparecer poco a poco. Es conveniente entregar con el producto una pagina gula de instalación y configuración en papel de manera que el usuario sepa el proceso que va a realizar al instalar la aplicación

Soporte técnico al instante, Cuando una empresa o cliente adquiere un paquete Informático suele obtener también un apoyo a distancia a la labor del usuario. Lo más común es el soporte telefónico y en los últimos tiempos se han popularizado las ayudas en red. Existen varios tipos:

- Telefónico: Se contrata como parte del mantenimiento
- Soporte mediante Internet: El mas extendido es el acceso a la Web del fabricante. Aquí se puede encontrar servicios para dientes y para cualquier navegante interesado.

- Soporte de acceso remoto: Es el menos extendido y se usa para evitar desplazamientos en casos en que la intervención técnica es sencilla de resolver.
- Soporto presencial: Es un tipo do soporto que ofrecen los fabricantes de aplicaciones a medida. Consisten en servicios de asistencia técnica a domicilio. Las empresas que ofrecen este soporte suelen ser grandes compañías con servicio técnico repartido por todo el territorio o empresas pequeñas con clientes cercanos geográficamente.

Bibliografía

Booch, Grady, El Lenguaje Unificado de Modelado. Addison Wesley Iberoamericana, 1999.

Sommerville, Ian. Ingeniería de Software. Prentice Hall. 2001.

Pressman Roger S. Ingeniería del Software, 5/E. Mc.Gaw-Hill. 2001

Jacobson, Ivar. Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley, 1992.

Larman, Craig UML y Patrones. Prentice Hall, 1999.

Object-Oriented Modeling and Design. J. Rumbaugh et al.Prentice- Hall,1991.

Oestereich, Bernd. Developing software with UML Object-oriented analysis and design in practice. Addison-Wesley, 2001.

Bruegge, Bernd. Ingeniería de software Orientada a objetos. Prentice Hall. 2000

UML Resource Center. Rational Software. http://www.rational.com/uml/