LAYERS **

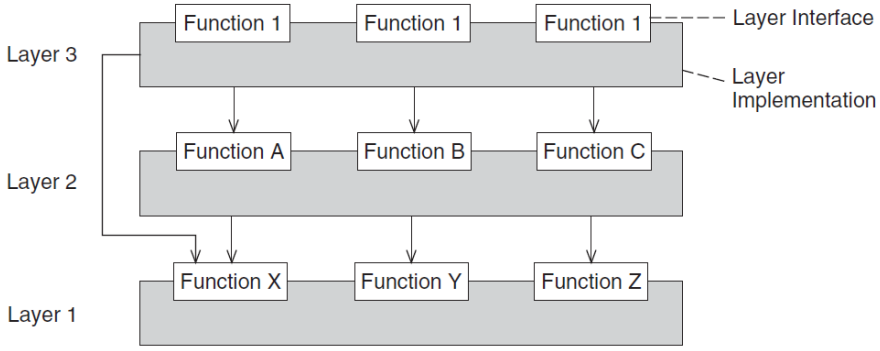
Al transformar un modelo de dominio (182) en una arquitectura de software técnico, o al realizar broker (237), la capa de acceso de base de datos
(438), microkernel (194), o half-sync/half-async *(359)*. . .
. . . Debemos apoyar el desarrollo independiente y la evolución de las diferentes partes del sistema.

___

Independientemente de las interacciones y de acoplamiento entre las diferentes partes de un sistema de software, hay una necesidad de desarrollar y evolucionar de forma independiente, por ejemplo, debido al tamaño y el tiempo de comercialización los requisitos del sistema. Sin embargo, sin una separación clara y razonada de las preocupaciones de la arquitectura del software del sistema, las interacciones entre las partes no pueden ser apoyados adecuadamente, ni puede su desarrollo independiente.

El desafío es encontrar un equilibrio entre un diseño que divide la aplicación en partes más significativas y tangibles que se pueden desarrollar y desplegar de forma independiente, pero no se pierda en una gran variedad de detalles para que la visión se pierde la arquitectura y las cuestiones operativas, tales como el rendimiento y escalabilidad no se tratan adecuadamente. Un diseño ad hoc monolítico no es una forma viable de resolver el desafío. A pesar de que permite la calidad de los aspectos del servicio que debe abordarse más directamente, es probable que resulte en una estructura espagueti que degrada las cualidades de desarrollo tales como comprensibilidad y facilidad de mantenimiento.
Por lo tanto:

Definir una o más capas para el software en desarrollo, con cada capa que tiene una responsabilidad distinta y específica.



Asigne la funcionalidad del sistema de las respectivas capas, y dejar que la funcionalidad de una capa particular sólo construir sobre la funcionalidad ofrecida por las mismas o más bajas capas. Proporcionar todas las capas con una interfaz que está separado de su aplicación, y dentro de cada programa de capa usando estas interfaces sólo cuando se accede a otras capas.

___

Una arquitectura de capas define una partición horizontal de la funcionalidad de un software de acuerdo a una propiedad (sub) de todo el sistema, de tal manera que cada grupo de funcionalidades está claramente encapsulado y puede evolucionar de forma independiente. Los criterios específicos de partición pueden ser definidos a lo largo de varias dimensiones, como abstracción, granularidad, distancia hardware, y la tasa de cambio. Por ejemplo, una estratificación que divide de una arquitectura en la presentación, la lógica de aplicación, y los datos persistentes sigue la dimensión abstracción. Una estratificación que presenta una capa de objeto de negocio cuyas entidades son utilizados por una capa de procesos de negocio sigue la granularidad de las dimensiones, mientras que uno que sugiere una

capa de abstracción del sistema operativo, una capa de protocolo de comunicaciones, y una capa con la funcionalidad de la aplicación sigue a la medida de separación del hardware. El uso de la tasa de cambio como criterio de estratificación separa las funciones que se desarrollan de forma independiente el uno del otro.

En la mayoría de las aplicaciones nos encontramos con múltiples dimensiones combinadas. Por ejemplo, la descomposición de una aplicación en la presentación, lógica de la aplicación, y las capas de datos persistentes es una estratificación según dos niveles de abstracción y la tasa de cambio. Las interfaces de usuario tienden a cambiar a una velocidad más alta que la lógica de aplicación, que evoluciona más rápido que los sistemas de datos, como las tablas de una base de datos relacional. Independientemente de la estratificación dimensiones de una aplicación sigue, cada capa utiliza la funcionalidad ofrecida por las capas inferiores a darse cuenta de su propia funcionalidad.

Un desafío clave es encontrar el número "correcto" de las capas. Muy pocos capas pueden no separar suficientemente los diferentes problemas en el sistema que puede evolucionar de forma independiente. Por el contrario, demasiadas capas pueden fragmentar una arquitectura de software en pedazos y sin una clara visión y alcance, lo que hace que sea difícil de evolucionar en absoluto. Además, las varias capas se definen, los más niveles de indirección deben cruzar en un flujo de control de extremo a extremo, lo cual puede introducir el rendimiento sanciones-especialmente cuando las capas son a distancia.

Normalmente, cada responsabilidad autónoma y coherente dentro de una capa se realiza como un objeto de dominio por separado, a la partición, la capa en partes tangibles que se pueden desarrollar y evolucionaron de forma independiente.

Dividir cada capa en una interfaz explícita (281) que publica las interfaces de los objetos de dominio cuya funcionalidad debe ser accesible por otras capas, y conectarlo con una implementación encapsulado (313), que da cuenta de esta funcionalidad. Esta separación de las preocupaciones minimiza el acoplamiento entre capas: cada capa sólo depende de interfaces de capa, lo que hace posible desarrollar una implementación de la capa con un impacto mínimo en otras capas, y también para proporcionar acceso remoto a una capa. Un puente (436) o un adaptador de objeto (438) apoya la separación de la interfaz explícita de una capa a partir de su aplicación encapsulado.

Control y de datos pueden fluir en ambas direcciones en los sistemas de capas. Por ejemplo, se intercambian datos entre capas adyacentes en las pilas de protocolos en capas, tales como TCP / IP o UDP / IP. Sin embargo, las capas define una dependencia baja a cíclico: capas inferiores no deben depender de las funciones proporcionadas por las capas superiores. Tal diseño evita la complejidad estructural accidental, y apoya el uso de capas más bajas en otras aplicaciones independientemente de las capas superiores. Por lo tanto, el flujo de control que se origina en el "fondo" de la pila es a menudo instigada por medio de un (399) Infraestructura de devolución de llamada basados en observador. Capas más bajas pueden pasar datos y solicitudes de servicio a través de las capas superiores notificaciones realizadas como comandos (412) o mensajes (420), sin llegar a depender de funciones específicas en sus interfaces.

Domain Object **

Al realizar un modelo de dominio (182), o su arquitectura técnica, en términos de capas (185), el modelo-vista-controlador (188), presentación de control abstracción (191), microkernel (194), reflexión (197), tuberías y filtros (200), repositorio compartido (202), o en la pizarra *(205). . . . .* Una de las

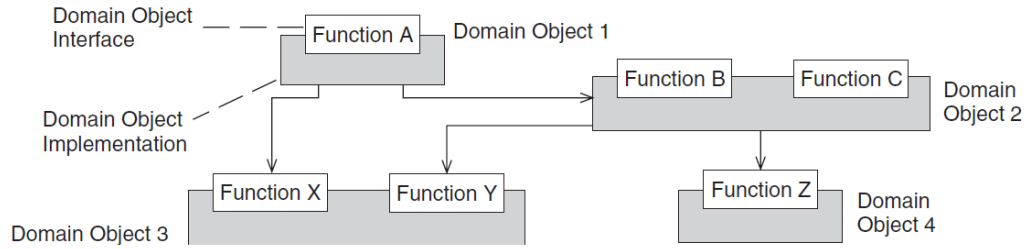
principales preocupaciones de todo el trabajo de diseño es desligar responsabilidades autónomas y de aplicación coherente entre sí.

___

Las partes que componen un sistema de software a menudo exponen relaciones de colaboración y de contención múltiples entre sí. Sin embargo, la implementación de esta funcionalidad interrelacionados sin cuidado puede resultar en un diseño con una alta complejidad estructural.

La separación de las preocupaciones es una propiedad clave de software bien diseñado. Cuanto más desacoplados son las diferentes partes de un sistema de software, mejor se pueden desarrollar y evolucionaron de forma independiente. El menor número de relaciones de las partes tienen el uno al otro, el más pequeño de la complejidad estructural de la arquitectura de software. El perdedor de las piezas se acoplan, mejor se pueden implementar en una red de ordenadores o compuestos en aplicaciones más grandes. En otras palabras, un reparto adecuado de un sistema de software evita la fragmentación de la arquitectura, y los desarrolladores mejor puede mantener, evolucionar y la razón de ello. Sin embargo, a pesar de la necesidad de una separación clara de las preocupaciones, la aplicación y la colaboración entre las diferentes partes de un sistema de software deben ser eficaces y eficientes para las calidades esenciales de funcionamiento, tales como el rendimiento, la gestión de errores, y la seguridad.

Por lo tanto:
Encapsular cada funcionalidad distinta de una aplicación en un edificio del bloque-un objeto de dominio autónomo.



Proporcionar a todos los objetos de dominio con una interfaz que está separado de su aplicación, y dentro de cada programa objeto de dominio sólo utilizando estas interfaces cuando se accede a otros objetos de dominio.

___

Objeto Dominio separa diferentes responsabilidades *funcionales* dentro de una aplicación de forma que cada funcionalidad es bien encapsulado y puede evolucionar de forma independiente. La división específica de las responsabilidades de una aplicación en objetos de dominio se basa en uno o varios criterios de granularidad. Un servicio de aplicación [acm01] es un objeto de dominio que encapsula un aspecto autónomo y función de negocio completo o la infraestructura de la aplicación, tales como la banca, la reserva del vuelo o servicio de registro [kaye03]. Un componente [vsw02] es un objeto de dominio que sea encapsula un bloque de construcción funcional, como un cálculo de impuesto sobre la renta o de una conversión de moneda, o una entidad de dominio como una cuenta de banco o un usuario. Un objeto de valor [ppr] [fow03a], un valor copiado (394), y un valor inmutable (396) son objetos de dominio pequeños cuya identidad se basa en su estado en lugar de su tipo, como una fecha, una tasa de cambio de divisas, o una cantidad de dinero. Un objeto de dominio también puede agregar otros objetos de dominio del mismo o menor granularidad. Por ejemplo, los servicios a menudo se crean a partir de componentes que utilizan objetos de valor.

Dividir cada objeto de dominio en una interfaz explícita (281) que exporta su funcionalidad y una implementación encapsulado (313), que da cuenta de la funcionalidad. Esta separación de interfaz y la implementación minimiza el acoplamiento entre dominios a objetos: cada objeto de dominio sólo depende de interfaces de objetos de dominio, pero no en la implementación de los objetos de dominio. Así, es posible realizar y desarrollar una implementación de objeto de dominio con un efecto mínimo sobre otros objetos de dominio. La interfaz explícita de un objeto de dominio define un contrato de propiedades operativas clave, como los aspectos de comportamiento de error y de seguridad, en la que otros objetos del dominio pueden confiar.

Hay varias opciones para la conexión de la interfaz explícita de un objeto de dominio con su aplicación encapsulada. Por ejemplo, Java y C # admiten el concepto de interfaz explícita en el núcleo del lenguaje, y las clases (implementaciones encapsuladas) pueden aplicar directamente. En otros lenguajes de tipo estático como C + +, una interfaz explícita se puede expresar como una clase base abstracta de la que se deriva de la aplicación explícita.

Un puente (436) o un adaptador de objeto (438) desacopla de forma explícita la interfaz explícita de un objeto de dominio de su aplicación encapsulado de modo que los dos se pueden variar independientemente. El grado de disociación entre la interfaz explícita de un objeto de dominio y su aplicación encapsulado depende de su granularidad y la probabilidad de cambio. El más pequeño de los objetos de dominio, por ejemplo, al realizar un objeto de valor o un valor inmutable, el desacoplamiento estricto menos beneficioso se convierte. Del mismo modo, cuanto más a menudo una aplicación encapsulada evoluciona, la mayor fuerza de la interfaz explícita deberá disociarse.

Las interfaces explícitas también permiten el acceso remoto a los objetos de dominio. Tenga en cuenta, sin embargo, que la comunicación remota es generalmente factible sólo para el dominio 'grande' objetos tales como servicios y componentes de grano grueso, pero no para los "pequeños" objetos de dominio como un objeto de valor. Cuanto más pequeño son los objetos de dominio, el más adverso es la razón de la creación de redes de arriba contra el tiempo de cálculo dentro del objeto de dominio, con sus correspondientes sanciones a los factores operativos de calidad, tales como rendimiento, disponibilidad y escalabilidad.

Objetos de dominio se asocia a menudo con una fábrica abstracta (525) o constructor (527) que permite a los clientes obtener acceso a su interfaz explícita y gestionar su vida de forma transparente. En plataformas como ccm [omg02], ejb [maha99], y. Neto [ram02], objetos de dominio se controlan mediante una configuración de componente declarativo (461) que especifica cómo su ciclo de vida, gestión de recursos y otros asuntos técnicos, como las transacciones y el registro debe ser manejado por su entorno de alojamiento. Un configurador de componentes (490) ayuda con la carga, en sustitución, (re) configuración y descarga de objetos de dominio en tiempo de ejecución.

Explicit Interface**

En el diseño de un (185), objetos de dominio (208), reactor (259), proactor (262), repetidor (298), compuesto (319), objeto activo (365), de control (412), interceptor (444) capas, cadena de responsabilidad (440), Puente (436), adaptador de objeto (438), visitante (447), decorador (453), estrateg (455), fachada de envoltura (459), observador (405) o un contenedor (488) disposición . . .

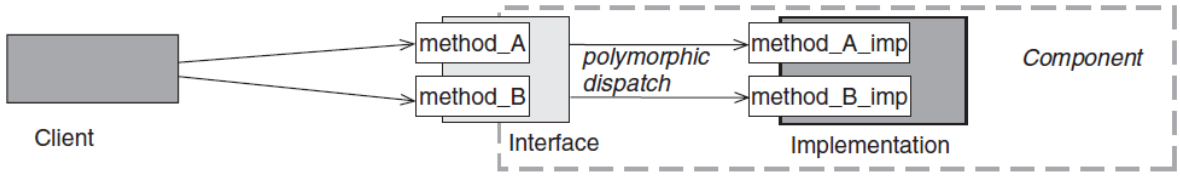
. . . Una de las principales preocupaciones de todos los trabajos de arquitectura de software es la expresión efectiva y adecuada de los interfaces de los componentes.

___

Un componente representa una unidad autónoma de la funcionalidad y la implementación con el uso de un protocolo publicado. Los clientes pueden utilizar como un bloque de construcción para proporcionar su propia funcionalidad. Acceso directo a la aplicación plena de componentes, sin embargo, haría dependiente clientes en partes internas de los componentes, lo que aumenta en última instancia, la aplicación de acoplamiento software interno.

Lo ideal sería que un cliente sólo debe depender de la interfaz publicada de un componente. Si esta interfaz se mantiene estable, las modificaciones en la implementación del componente no deberían afectar a sus clientes. La encapsulación de componentes dentro de las clases concretas ordinarias es práctico: estas interfaces de clase siempre están ligados a sus implementaciones. Incluso con las clases abstractas, la inclusión típico de una aplicación parcial es más vinculante que es apropiado para el acoplamiento débil y la estabilidad. Ubicación independencia es una preocupación adicional: los clientes de un componente pueden residir en espacios de direcciones remotas, y la ubicación de un componente pueden cambiar en tiempo de ejecución, por lo que, por tanto, las dependencias del cliente en la ubicación de un componente debe ser evitado. Finalmente, el método ofrecido por un componente debe ser significativa para los clientes y apoyar su eficaz y correcta de uso, especialmente en implementaciones distribuidas o concurrentes.

Por lo tanto:
Separar la interfaz declarada de un componente de su aplicación. Exportar la interfaz para los clientes del componente, pero mantendrá su implementación privada y la ubicación-transparente para el cliente.



Una llamada desde el cliente a través de esta interfaz explícita será remitida al componente, pero el código del cliente dependerá sólo de la interfaz y no en la aplicación. Una interfaz explícita está asociado a un contrato [mey97] que los clientes *deben* seguir para utilizar un componente tan correctamente. Este contrato incluye las operaciones que ofrece el componente, el protocolo para llamar a las operaciones, y cualesquiera otras limitaciones y la información que los clientes deben saber para utilizar el componente de forma correcta y eficaz.

___
Una interfaz explícita impone una estricta separación de la interfaz del componente de su aplicación concreta, que separa los problemas de uso de los componentes de la realización concreta y detalles de la ubicación. Esta separación también permite la modificación transparente de implementaciones de componentes de forma independiente de los clientes que lo utilizan, siempre y cuando el contrato definido por las interfaces sigue siendo estable.

Varios patrones ayudan en la estructuración de interfaz explícita de un componente. Una interfaz de extensión (284) soporta la partición de una interfaz explícita en múltiples interfaces más pequeñas, una para cada función del componente. Interfaz de extensión también permite la extensión del componente con nuevas interfaces específicas de la función. En general, una interfaz de extensión es compatible con la evolución de interfaz y reducir al mínimo el efecto de esta evolución en los clientes del componente.

Un proxy (290) ayuda a encapsular tareas específicas de mantenimiento de la casa de asociados con la invocación de un componente. Por ejemplo, se puede transformar una llamada a un método en un mensaje que se puede enviar a través de la red para la implementación del componente, cargarlo desde la base de datos en el primer acceso, o caché estado inmutable para acceso de cliente eficiente. Un delegado de la empresa (292), por el contrario, es más útil en entornos dinámicos distribuir, que normalmente requieren de una serie de tareas de infraestructura que se realizarán en el acceso al componente. Por ejemplo, antes de que un componente puede ser invocado, su aplicación debe primero ser localizado y una conexión con su aplicación debe ser establecido. Un delegado de negocio puede ejecutar estas tareas de forma transparente para los clientes cuando invocan un método en el componente. Una fachada (294) protege a los clientes de la estructura interna del componente, que puede consistir en partes aún más pequeñas. Se proporciona un único punto de entrada definido en el componente, que permite que la estructura del componente a ser variada sin efectos sobre sus clientes.

Un tema clave al especificar una interfaz explícita es la calidad operativa: los clientes deben ser capaces de utilizar el componente con eficacia y correctamente. Diseñar una interfaz explícita como una interfaz segura para subprocesos (384) serializa el acceso a un componente en los escenarios de uso concurrente, manteniendo el bloqueo sobrecarga al mínimo y, en el caso de los bloqueos no recursivos, evitando la auto-bloqueo debe un componente invocar métodos sobre sí misma. Autorización (351) garantiza el acceso seguro a las funciones del componente. Un método combinado (296) representa una serie de llamadas a métodos en el componente que siempre se llamó y en un orden específico, lo que hace que una interfaz explícita más expresiva, ya que refleja el uso de componentes comunes. Métodos de fábrica (529) y los métodos de eliminación (531) permiten a los clientes crear y disponer de un componente sin depender ni su estructura interna o de los procesos de construcción y destrucción.

Si una interfaz representa una estructura de agregados tales como una colección, los clientes pueden necesitar acceso a los elementos o si desea ejecutar acciones sobre ellos. Un iterador (298) permite a los clientes atraviesan estos elementos uno a la vez sin romper la encapsulación del componente. Un método de lotes (302) es similar en intención de un iterador, pero envía o devuelve varios elementos en cada invocación, lo cual es beneficioso para los sistemas distribuidos y concurrentes, debido a la creación de redes y los gastos generales de sincronización se reduce al mínimo. Un método de enumeración (300) ayuda a ejecutar una acción específica en cada elemento sin necesidad de la persona que llama para gestionar el recorrido de forma explícita, lo que minimiza la sobrecarga de sincronización en escenarios de uso concurrentes.

Los parámetros y resultados de una invocación de una interfaz explícita se pueden encapsular en objetos de transferencia de datos (418), lo que evita los clientes o el componente de tener que depender de representaciones de datos concretos. Si los clientes necesitan tener acceso a el estado interno del componente, se puede devolver como un recuerdo (414) para mantener la encapsulación. Si el componente de las necesidades de información específica del cliente para ejecutar sus servicios, se puede pasar como un objeto de contexto (416) para el componente.

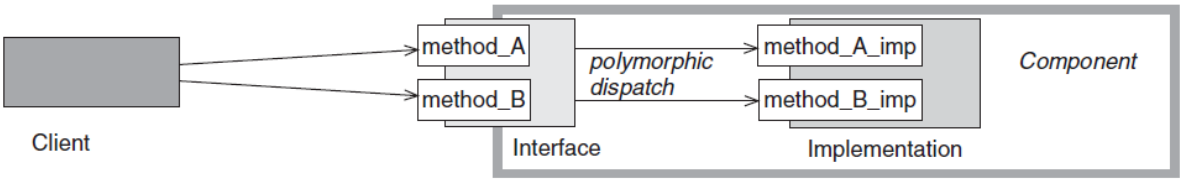
Encapsulated Implementation**

En el desarrollo de un (185) Arquitectura, objetos de dominio de una aplicación (208), un puente (436) disposición, o en general, al diseñar los componentes para un sistema basado en componentes *capas. . .* . . . Una decisión importante consiste en la realización de las implementaciones de los componentes contra interfaces de los componentes.

___

Un componente ofrece sus servicios a través de interfaces que definen los protocolos de uso, funcionalidad publicada, y la calidad de las propiedades del servicio. Sin embargo, una interfaz es sólo una promesa: un componente debe proporcionar satisfacción. Así, una implementación del componente es a menudo sometido a supuestos, consideraciones y limitaciones que no pueden ser expuestos a través de su interfaz.

Independientemente del contrato definido por la interfaz de un componente, su realización está expuesta a las limitaciones y requisitos que son de poco interés para los usuarios que lo componen, sino que son cruciales en el cumplimiento del contrato. Por ejemplo, puede que tenga que estar preparado para la implementación distribuida sin disminuir sus propiedades de rendimiento y rendimiento del componente. O, dependiendo de su uso y despliegue escenarios concretos, diferentes algoritmos, funcionalidades adicionales, o propiedades adicionales de la calidad puede ser necesaria, por ejemplo, los diferentes algoritmos de cálculo de impuestos, una actividad empresarial específica del usuario, o medidas de seguridad más fuertes. Por último, casi todos los componentes están sujetos t evolución, su aplicación puede cambiar con el tiempo. Sin embargo, los clientes deben estar protegidos de todos estos aspectos-que sólo están interesados en el contrato, no en la forma en que se ha cumplido.
Por lo tanto:
Asegúrese de que todos los detalles de la implementación de los componentes permanecen ocultos detrás de sus interfaces para proteger a los clientes de las opciones de representación que pueden cambiar durante el tiempo de vida de una aplicación, o dependen de la implementación específica del componente.



El cliente de un componente no puede asumir la programación más se expone a través de su interfaz oficial, que mejora la habilidad implementación del componente de cambiar la implementación sin clientes rompiendo.

___
Una implementación del componente que respete los límites definidos por las interfaces asegura que la dependencia de sus clientes está en su interfaz, la totalidad de su interfaz, y nada más que su interfaz. La aplicación encapsulado es libre de evolucionar, preservando al mismo tiempo la estabilidad de sus clientes. Los desarrolladores de código de cliente todavía se presentan con una interfaz sencilla y estable para su uso.

Los sellos de la interfaz de límites de aplicación del componente de su medio ambiente, y viceversa, pero es posible que el componente tiene una dependencia en algunas características de su entorno de la llamada. Para evitar la introducción de una dependencia inversa, pasar objetos de contexto (416) de la persona que llama para el componente cuando se requiere dicha información o comportamiento. Un componente que también tenga que perder el estado depende de la implementación del cliente para su uso posterior en relación con el componente, por ejemplo, una posición en un recorrido o un controlador de devolución de llamada para un evento registrado de interés. En este caso, preservar la encapsulación del componente mediante la devolución de un recuerdo (414).

En general, una implementación encapsulado debe proporcionar una representación de software bien encapsulado de responsabilidades funcionales específicos del componente. Patrones específicos de dominio pueden apoyar la creación de esta representación, como los de atención de la salud, finanzas corporativas, telecomunicaciones y aplicaciones de transporte público [fow97] [ris01] [plopd1] [plopd2] [plopd3] [plopd4] [plopd5].

A veces, la representación de las funciones del componente también puede definirse sobre la base de los patrones más generales que no están vinculados a un dominio específico (aplicación). Por ejemplo, componentes que representan una jerarquía de elementos se pueden realizar utilizando toda una parte (317) o compuesto (319) de diseño, depende de cómo distinta o uniforme los elementos de la jerarquía son. Si la funcionalidad del componente se centra en una máquina de estados, su realización puede estar basado en un objeto de estados (467) o los métodos de los estados (469) de diseño, dependiendo del tamaño de la máquina de estado y la cantidad de datos y el contexto información compartida entre los estados. Si la responsabilidad del componente es interpretar archivos estructurados o frases de un idioma determinado, como por ejemplo en un analizador, un (442) Diseño intérprete podría ser considerada como su estructura fundamental.

Una de las principales preocupaciones de todas las implementaciones encapsulados está cumpliendo con las cualidades de funcionamiento especificados en el contrato del componente: el rendimiento, escalabilidad, etc. Proporcionar la estructura "derecho" y el comportamiento simplemente no es suficiente para asegurar la facilidad de uso de un componente y la aceptación. Dos técnicas que ayudan en la consecución de las cualidades operativas clave son la distribución y concurrencia.

Implementar una aplicación encapsulada a múltiples hosts en un sistema distribuido permite a un componente de aprovechar los recursos disponibles en toda la red, y no sólo de los que están disponibles en un único nodo de la red. Los más recursos disponibles, mejor calidad operativa del componente. ¿Qué cualidades en particular puede ser apoyado por la implementación distribuida depende de la partición componente elegido. Maestro-esclavo (321), medio-objeto plus de protocolo (324), y el grupo de componentes replicado (326) ofrecen diferentes ventajas y desventajas para el sostenimiento de rendimiento, disponibilidad, escalabilidad, tolerancia a fallos y la precisión computacional.

A efectos de aplicación concurrentes encapsuladas las características de utilización de un componente específicamente su desempeño y rendimiento, porque múltiples solicitudes de los clientes pueden ser manejados y procesados simultáneamente. Un objeto activo (365) disposición apoya la implementación de los componentes de su propio conjunto de temas, mientras que un objeto de monitor (368) ayuda en la realización de los componentes que están colocadas dentro de sus subprocesos de cliente. Half-sync/half-async (359) y / líder de seguidores (362) configuraciones son las más convenientes para los componentes que la red proceso de i / o.

Sin embargo, todos los modelos de despliegue y de concurrencia encima vienen con ciertos costos, que se deben principalmente a la funcionalidad duplicada, aumento del uso de los recursos, la aplicación más compleja, y la coordinación entre las partes concurrentes de la aplicación encapsulada u otros componentes distribuidos. Antes de la introducción de cualquiera de estos modelos, es importante evaluar si sus costos son mayores que sus beneficios.

Otra de las principales preocupaciones para las implementaciones de casi todos encapsulados es el apoyo a la evolución, la extensión y la adaptación. Estas cualidades de desarrollo permiten el uso eficaz

de los componentes dentro de diferentes implementaciones de aplicaciones y variantes, y también su reutilización dentro de aplicaciones completamente diferentes.

Estrategias (455) y los métodos de la plantilla (453), por ejemplo, apoyar la separación de la variante de la conducta invariable, la estrategia mediante el uso de la delegación, y el método de la plantilla por herencia. En contraste, el visitante (447) permite que la funcionalidad que se añade a una implementación encapsulado que no se previó durante su desarrollo original. El flujo de control dentro de un componente se puede ampliar mediante interceptores (444), decoradores (472) y, si es c + + se utiliza para realizar la aplicación encapsulada, ejecutar-en torno a los objetos (451). Los interceptores pueden inyectar comportamiento fuera de banda en el flujo de control de una función, mientras que decoradores ayudan en envolver una función con comportamiento especializado que ha de ser ejecutado antes o después de una función. Un objeto ejecutar-en torno a es similar en este sentido a un decorador: permite una funcionalidad adicional que se debe ejecutar antes o después de una secuencia de instrucciones de una manera segura excepción, que la convierte en una c + + lenguaje preferido para la adquisición de recursos y la liberación. Por último, los adaptadores de objetos (438) y las fachadas de envoltura (459) apoyan la integración de una aplicación encapsulada con su entorno mediante la adaptación de las *interfaces* provistas de componentes, bibliotecas y sistemas operativos a los *esperados* o *requerida* por la aplicación encapsulada. La diferencia entre las dos opciones de diseño es que el adaptador de objeto no oculta las interfaces adaptadas, que siguen siendo accesibles en la aplicación encapsulada, mientras que en un arreglo envoltorio fachada estas interfaces están completamente sellados.

Si bien es necesario un cierto grado de flexibilidad para utilizar una aplicación encapsulada con eficacia en aplicaciones concretas, demasiada flexibilidad podría resultar en lo contrario: un componente que es tan flexible que no sirve de nada en absoluto [BUS03]. Por tanto, es importante que sólo la variabilidad obligatoria se apoya en una aplicación encapsulada, no toda la variabilidad 'agradable a tener ". La variabilidad obligatoria para un componente se puede identificar con la ayuda de los métodos adecuados, tales como *análisis de la aplicación abierta y el diseño* [kllm95], *el análisis de concordancia / variabilidad* [cope98], o *el modelado* [czei02] *característica.*