Concepto

- Los patrones de diseño para el desarrollo de software representan una herramienta para facilitar la producción de aplicaciones más robustos y más reusables.
- Se intenta plasmar los conceptos que se encuentran frecuentemente en las aplicaciones en algún tipo de *código genérico*.
- Un concepto muy parecido a los patrones de diseño se encuentra en la matemática y en la teoría de los algoritmos, por ejemplo:

Matemática

técnicas de pruebas matemáticas:

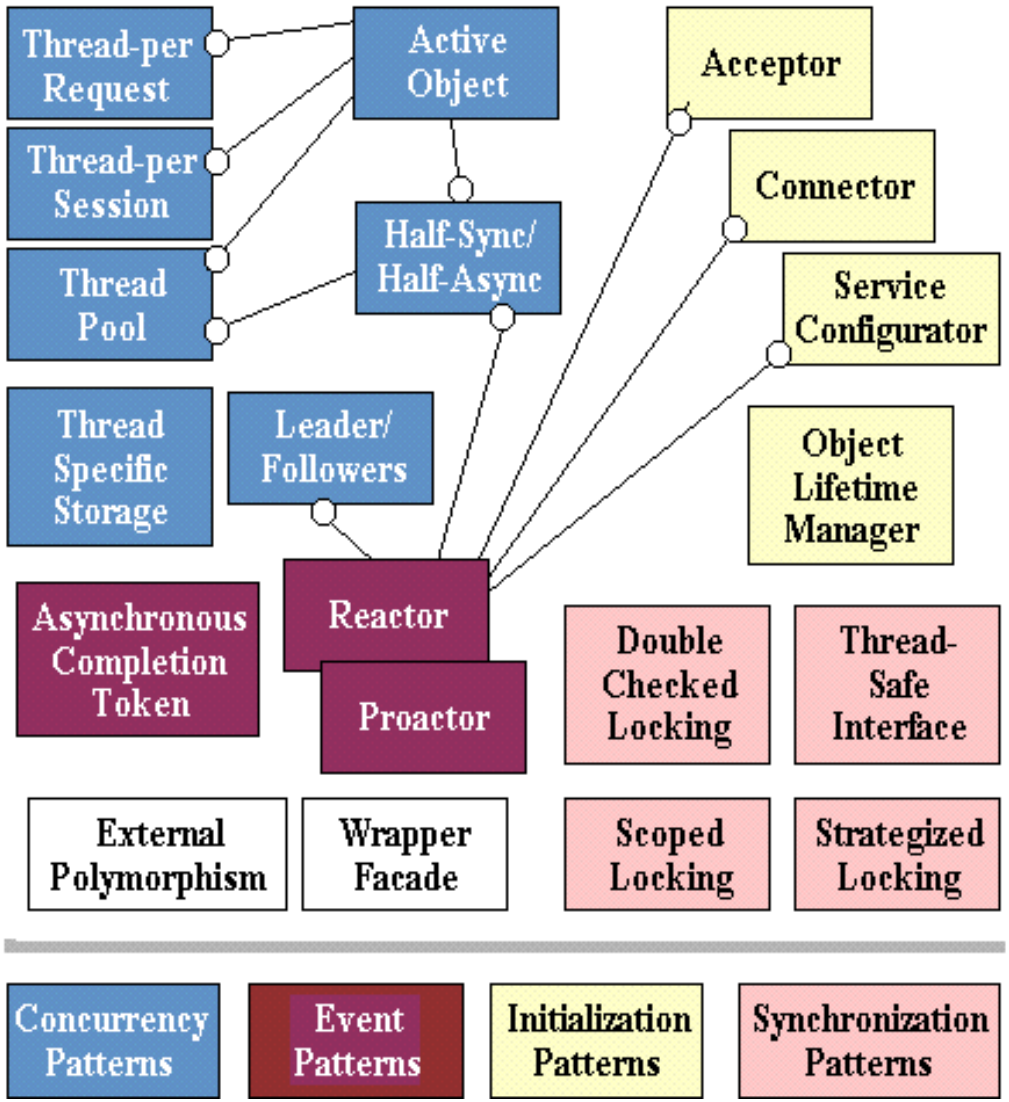
- comprobación directa
- inducción
- contradicción
- contra-ejemplo
- comprobación indirecta
- diagonalización
- reducción
- y más

Algoritmia

paradigmas de desarrollo de algoritmos:

- iteración
- recursión
- búsqueda exhaustiva
- búsqueda binaria
- divide–y–vencerás
- ramificación-y–poda
- barrido
- perturbación
- amortización
- y más

Patrones



http://www.cs.wustl.edu/~schmidt/ACE-papers.html

Patrones

A continuación veremos unos patrones de diseño útiles para la programación concurrente.

- Reactor
- Proactor
- Ficha de terminación asíncrona
- Guardián
- Aviso de hecho (y su uso para el Singleton)

Crítica

- Patrones de diseño no son *el-no-va-más*.
- Muchas veces solamente expresan propiedades que deben estar ya incorporado en el propio lenguaje a alto nivel.
- Sirven como *lenguaje* de comunicación entre programadores, pero hay que tener cuidado que se habla con la misma semántica.
- A veces están demasiado ligados a una implementación en concreta y su uso empuja decisiones a nivel de implementación al nivel de diseño o incluso analysis (fallo típico de un *mero* programador).

Uso

Se usa cuando una aplicación

- que gestióna eventos
- debe reaccionar a varias peticiones cuasi simultaneamente,
- pero las procesa de modo síncrono y en el orden de llegada.

Ejemplos:

- servidores con multiples clientes
- interfaces al usuario con varias fuentes de eventos
- servicios de transacciones
- *centralita*

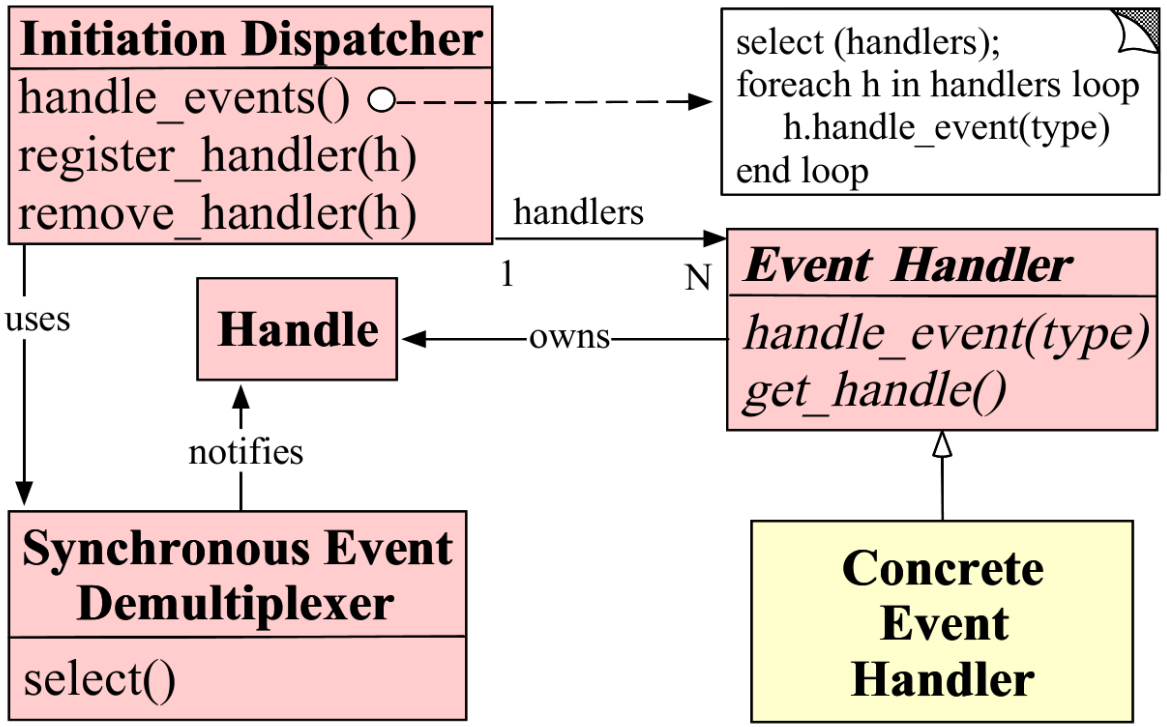
Comportamiento exigido

- La aplicación no debe bloquear innecesariamente otras peticiones mientras se está gestionando una petición.
- Debe ser fácil incorporar nuevos tipos de eventos y peticiones.
- La sincronización debe ser escondida para facilitar la implementación de la aplicación.

Posible solución

- Se espera en un bucle central a todos los eventos que pueden llegar.
- Una vez recibido un evento se traslada su procesamiento a un gestor específico de dicho tipo de evento.
- El reactor permite añadir/quitar gestores para eventos.

Reactor: posible diagrama



(image taken from: D.C. Schmidt, Reactor, 1995)

Detalles de la implementación

- Bajo Unix y (parcialmente) bajo Windows se puede aprovechar de la función `select()` para el bucle central.
- Hay que tener cuidado que los eventos en espera tengan posibilidad de llegar al actor por lo menos con espera finita garantizada.
- Si los gestores de eventos son procesos independientes hay que evitar posibles interbloqueos o estados erroneos si varios gestores trabajan con un estado común.
- Se puede aprovechar del propio mecanismo de gestionar eventos para lanzar eventos que provoquen que el propio *reactor* cambie su estado.
- Java no dispone de un mecanismo apropriado para emular el `select()` de Unix (hay que usar programación multi-hilo con sincronización).

Uso

Se usa cuando una aplicación

- que gestióna eventos
- debe actuar en respuesta a varias peticiones casi simultaneamente y
- debe procesar los eventos de modo asíncrono notificando la terminación adecuadamente.

Ejemplos:

- servidores para la Red
- interfaces al usuario para tratar componentes con largos tiempos de cálculo
- *contestador automático*

Comportamiento exigido

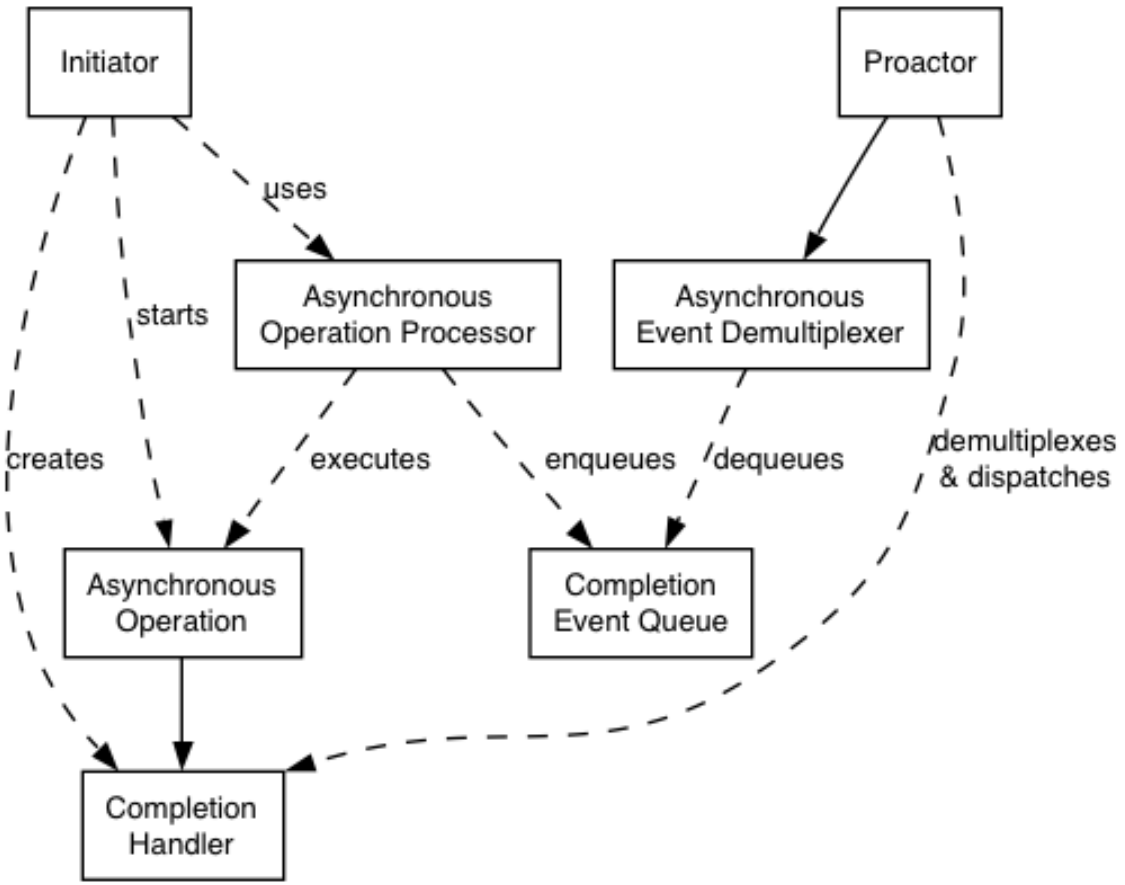
(igual como en el caso del reactor)

- La aplicación no debe bloquear innecesariamente otras peticiones mientras se está gestionando una petición.
- Debe ser fácil incorporar nuevos tipos de eventos y peticiones.
- La sincronización debe ser escondida para facilitar la implementación de la aplicación.

Posible solución

- Se divide la aplicación en dos partes: operaciones de larga duración (que se ejecutan asíncronamente) y administradores de eventos de terminación para dichas operaciones.
- Con un iniciador se lanza cuando haga falta la operación compleja.
- Las notificaciones de terminación se almacena en una cola de eventos que a su vez el administrador está vaciando para notificar la aplicación de la terminación del trabajo iniciado.
- El proactor permite añadir/quitar gestores para operaciones y administradores.

Proactor: posible diagrama



(image taken from: Boost library, 2012)

Detalles de la implementación

- Muchas veces basta con un solo proactor en una aplicación que se puede implementar a su vez como *singleton*.
- Se usa varios proactores en caso de diferentes prioridades (de sus colas de eventos de terminación).
- Se puede realizar un bucle de iniciación/terminación hasta que algún tipo de terminación se haya producido (por ejemplo transpaso de ficheros en bloques y cada bloque de modo asíncrono.
- La operación asíncrona puede ser una operación del propio sistema operativo.

Uso

Se usa cuando una aplicación

- que gestióna eventos
- debe actuar en respuesta a sus propias peticiones
- de modo asíncrono después de ser notificado de la terminación del procesamiento de la petición.

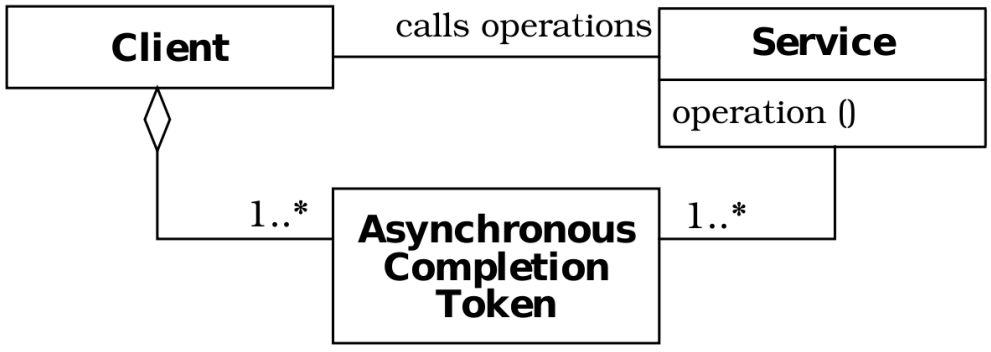
Ejemplos:

- interacción compleja en un escenario de commercio electrónico (relleno de formularios, suscripción a servicios)
- interfaces al usuario con diálogos no bloqueantes
- *contestador automático*

Comportamiento exigido

- Se quiere separar el procesamiento de respuestas a un servicio.
- Se quiere facilitar un servicio a muchos clientes sin mantener el estado del cliente en el servidor.

Posible solución



`http://www.cs.wustl.edu/~schmidt/ACE-papers.html`

- La aplicación manda con su petición una ficha indicando como hay que procesar después de haber recibido un evento de terminación de la petición.
- La notificación de terminación incluye la ficha original.

Detalles de la implementación

- Las fichas suelen incorporar una identificación.
- Las fichas pueden contener directamente punteros a datos o funciones.
- En un entorno más heterógeno se puede aprovechar de objetos distribuidos (CORBA).
- Hay que tomar medidas de seguridad para evitar el proceso de fichas no-deseados.
- Hay que tomar medidas para el caso de perder eventos de terminación.

Uso

Se usa cuando una aplicación

- contiene procesos (hilos) que se ejecutan concurrentemente y
- hay que proteger bloques de código con un punto de entrada pero varios puntos de salida
- para que no entren varios hilos a la vez.

Ejemplos:

- cualquier tipo de protección de secciones críticas

Comportamiento exigido

- Se quiere que un proceso queda bloqueado si otro proceso ya ha entrado en la sección crítica, es decir, ha obtenido la llave exclusiva de dicha sección.
- Se quiere que independientemente del método usado para salir de la sección crítica (por ejemplo uso de `return`, `break` etc.) se devuelve la llave exclusiva para la región.

Posible solución

- Se inicializa la sección crítica con un objeto de guardia que intenta obtener una llave exclusiva.
- Se aprovecha de la llamada automática de destructores para librar la sección crítica, es decir, devolver la llave.

Detalles de la implementación I

- Java proporciona el guardián directamente en el lenguaje: métodos y bloques sincronizados (`synchronized`).
- Hay que prevenir auto-bloqueo en caso de llamadas recursivas.
- Hay que tener cuidado con interrupciones forzadas que circundan el flujo de control normal.
- Porque el guardián no está usado en la sección crítica, el compilador puede efectuar ciertos mensajes de alerta y — en el caso peor — un optimizador puede llegar a tal extremo eliminando el objeto.

Detalles de la implementación II

- Para facilitar la implementación de un guardián en diferentes entornos (incluyendo situaciones secuenciales donde el guardián efectivamente no hace nada) se puede aprovechar de estrategias de polimorfismo o de codificación con plantillas para flexibilizar el guardián (el patrón así cambiado se llama a veces: *strategized locking*).

Uso

Se usa cuando una aplicación

- usa objetos (clases) que necesitan una inicialización única y exclusiva (patrón *Singleton*)
- que no se quiere realizar siempre
- sino solamente en caso de necesidad explícita y
- que puede ser realizada por cualquier hilo que va a usar el objeto por primera vez.

Ejemplos:

- construcción de singletons

Comportamiento exigido

- Se quiere un trabajo mínimo en el caso que la inicialización ya se ha llevado a cabo.
- Se quiere que cualquier hilo puede realizar la inicialización.
- Se quiere inicializar solamente en caso de necesidad real.

Posible solución

- Se usa un guardián para obtener exclusión mutua.
- Se comprueba dos veces si la inicialización ya se ha llevado a cabo: una vez antes de obtener la llave y una vez después de haber obtenido la llave.

Detalles de la implementación

- Hay que marcar la bandera que marca si la inicialización está realizada como volátil (`volatile`) para evitar posibles optimizaciones del compilador.
- El acceso a la bandera tiene que ser atómico.

Hay que estar alerta a problemas y cosas nuevas...

- ... mira el artículo de Meyers...
- Scott Meyers and Andrei Alexandrescu. C++ and The Perils of Double-Checked Locking. Dr. Dobb's, The World of Software Development. July 01, 2004
  (versión en PDF en página del curso)
- que demuestra los problemas del patrón en C++03

... que ofrecen soluciones

Pero ahora hay C++11, y con eso funciona lo siguiente, dado que la construcción de objetos estáticos está garantizado de ser seguro con hilos:

```
class Foo
{
public:
    static Foo& instance( void )
    {
        static Foo s_instance;
        return s_instance;
    }
};
```

Más patrones para la concurrencia

- Aceptor–Conector
- Objetos activos
- Monitor
- Mitad-síncrono, mitad-asíncrono
- Líder–y–Seguidores
- Interfaz segura para multi-hilos

Uso

Se usa cuando una aplicación

- necesita establecer una conexión entre una pareja de servicios (por ejemplo, ordenadores en una red)
- donde el servicio sea transparente a las capas más altas de la aplicación
- y el conocimiento de los detalles de la conexión (activo, pasivo, protocolo) no son necesarios para la aplicación.

Ejemplos:

- los super-servicios de unix (`inetd`)
- usando `http` para realizar operaciones (CLI)

Comportamiento exigido

- Se quiere esconder los detalles de la conexión entre dos puntos de comunicación.
- Se quiere un mecanismo flexible en la capa baja para responder eficientemente a las necesidades de aplicaciones para que se puedan jugar papeles como servidor, cliente o ambos en modo pasivo o activo.
- Se quiere la posibilidad de cambiar, modificar, o añadir servicios o sus implementaciones sin que dichas modificaciones afecten directamente a la aplicación.

Posible solución

- Se separa el establecimiento de la conexión y su inicialización de la funcionalidad de la pareja de servicios (*peer services*), es decir, se usa una capa de transporte y una capa de servicios.
- Se divide cada pareja que constitúe una conexión en una parte llamada aceptor y otra parte llamada conector.
- La parte aceptora se comporta pasiva esperando a la parte conectora que inicia activamente la conexión.
- Una vez establecida la conexión los servicios de la aplicación usan la capa de transporte de modo transparente.

Detalles de la implementación I

- Muchas veces se implementa un servicio par–en–par (*peer–to–peer*) donde la capa de transporte ofrece una pareja de conexiones que se puede utilizar independientemente en la capa de servicios, normalmente una línea para escribir y otra para recibir.
- La inicialización de la capa de transporte se puede llevar a cabo de modo síncrono o asíncrono, es decir, la capa de servicios queda bloqueada hasta que se haya establecido la conexión o se usa un mecanismo de notificación para avisar a la capa de servicios del establecimiento de la conexión.

Detalles de la implementación II

- Es recomendado de usar el modo síncrono solamente cuando: el retardo esperado para establecer la conexión es corto o la aplicación no puede avanzar mientras no tenga la conexión disponible.
- Muchas veces el sistema operativo da soporte para implementar este patrón, por ejemplo, conexiones mediante sockets.
- Se puede aprovechar de la misma capa de transporte para dar servicio a varias aplicaciones a la vez.

Uso

Se usa cuando una aplicación

- usa varios hilos y objetos
- donde cada hilo puede realizar llamadas a métodos de varios objetos que a su vez se ejecutan en hilos separados.

Ejemplos:

- comportamiento de camarero y cocina en un restaurante

Comportamiento exigido

- Se quiere una alta disponibilidad de los métodos de un objeto (sobre todo cuando no se espera resultados inmediatos, por ejemplo, mandar mensajes).
- Se quiere que la sincronización necesaria para involucrar los métodos de un objeto sea lo más transparente que sea posible.
- Se quiere una explotación transparente del paralelismo disponible sin programar explicitamente planificadores en la aplicación.

Posible solución

- Para cada objeto se separa la llamada a un método de la ejecución del código (es decir, se usa el patrón *proxy*).
- La llamada a un método (que se ejecuta en el hilo del cliente) solamente añade un mensaje a la lista de acciones pendientes del objeto.
- El objeto ejecuta con la ayuda de un planificador correspondiente las acciones en la lista.
- La ejecución de las tareas no sigue necesariamente el orden de pedidos sino depende de las decisiones del planificador.
- La sincronización entre el cliente y el objeto se realiza básicamente sobre el acceso a la lista de acciones pendientes.

Detalles de la implementación

- Para devolver resultados existen varias estrategias: bloqueo de la llamada en el proxy, notificación con un mensaje (interrupción), uso del patrón futuro (se deposita el objeto de retorno a la disposición del cliente).
- Debido al trabajo adicional el patrón es más conveniente para objetos gruesos, es decir, dondo el tiempo de cálculo de sus métodos por la frecuencia de sus llamadas es largo.
- Se tiene que tomar la decisión apropriada: uso de objetos activos o uso de monitores.
- Se puede incorporar temporizadores para abordar (o tomar otro tipo de decisiones) cuando una tarea no se realiza en un tiempo máximo establecido.

Uso

Se usa cuando una aplicación

- consiste en varios hilos
- que actuan sobre el mismo objeto de modo concurrente

Ejemplos:

- colas de pedido y colas de espera en un restaurante tipo comida rápida

Comportamiento exigido

- Se protege los objetos así que cada hilo accediendo el objeto vea el estado apropiado del objeto para realizar su acción.
- Se quiere evitar llamadas explícitas a semáforos.
- Se quiere facilitar la posibilidad que un hilo bloqueado deje el acceso exclusivo al objeto para que otros hilos puedan tomar el mando (aún el hilo queda a la espera de re-tomar el objeto de nuevo).
- Si un hilo suelta temporalmente el objeto, este debe estar en un estado adecuado para su uso en otros hilos.

Posible solución

- Se permite el acceso al objeto solamente con métodos sincronizados.
- Dichos métodos sincronizados aprovechan de una sola llave (llave del monitor) para encadenar todos los accesos.
- Un hilo que ya ha obtenido la llave del monitor puede acceder libremente los demás métodos.
- Un hilo reestablece en caso que puede soltar el objeto un estado de la invariante del objeto y se adjunta en una cola de espera para obtener acceso de nuevo.
- El monitor mantiene un conjunto de condiciones que deciden los casos en los cuales se puede soltar el objeto (o reanuar el trabajo para un hilo esperando).

Detalles de la implementación

- Los objetos de Java implícitamente usan un monitor para administrar las llamadas a métodos sincronizados.
- Hay que tener mucho cuidado durante la implementación de los estados invariantes que permiten soltar el monitor temporalmente a la reanuación del trabajo cuando se ha cumplido la condición necesaria.
- Hay que prevenir el posible bloqueo que se da por llamadas intercaladas a monitores de diferentes objetos: se suelta solamente el objeto de más alto nivel y el hilo se queda esperando en la cola de espera (un fallo común en Java).

Uso

Se usa cuando una aplicación

- tiene que procesar servicios síncronos y asíncronos a la vez
- que se comunican entre ellos

Ejemplos:

- administración de dispositivos controlados por interrupciones
- unir capas de implementación de aplicaciones que a nivel bajo trabajan en forma asíncrono pero que hacia ofrecen llamadas síncronas a nivel alto (por ejemplo, `read/write` operaciones a trajetas de red)
- organización de mesas en restaurantes con un camarero de recepción

Comportamiento exigido

- se quiere ofrecer una interfaz síncrona a aplicaciones que lo desean
- se quiere mantener la capa asíncrona para aplicaciones con altas prestaciones (por ejemplo, ejecución en tiempo real)

Posible solución

- se separa el servicio en dos capas que están unidos por un mecanismo de colas
- los servicios asíncronos pueden acceder las colas cuando lo necesitan con la posibilidad que se bloquea un servicio síncrono mientras tanto

Detalles de la implementación

- hay que evitar desbordamiento de las colas, por ejemplo, descartar contenido en ciertas ocaciones, es decir, hay que implementar un control de flujo adecuado para la aplicación
- se puede aprovechar de los patrones *objetos activos* o *monitores* para realizar las colas
- para evitar copias de datos innesesarias se puede usar memoria compartida para los datos de las colas, solamente el control de flujo está separado

Uso

Se usa cuando una aplicación

- tiene que reaccionar a varios eventos a la vez y
- no es posible o conveniente inicializar cada vez un hilo para cada evento

Ejemplos:

- procesamiento de transacciones en tiempo real
- colas de taxis en aeropuertos

Comportamiento exigido

- se quiere una distribución rápida de los eventos a hilos ya esperando
- se quiere garantizar acceso con exclusión mutua a los eventos

Posible solución

- se usa un conjunto de hilos organizados en una cola
- el hilo al frente de la cola (llamado líder) procesa el siguiente evento
- pero transforma primero el siguiente hilo en la cola en nuevo líder
- cuando el hilo ha terminado su trabajo se añade de nuevo a la cola

Detalles de la implementación

- se los eventos llegan más rápido que se pueden consumir con la cola de hilos, hay que tomar medidas apropriadas (por ejemplo, manejar los eventos en una cola, descartar eventos etc.)
- para aumentar la eficiencia de la implementación se puede implementar la cola de hilos esperando como un pila
- el acceso a la cola de seguidores tiene que ser eficiente y robusto

Uso

Se usa cuando una aplicación

- usa muchos hilos que trabajan con los mismos objetos
- y se quiere minimizar el trabajo adicional para obtener y devolver la llave que permite acceso en modo exclusivo.

Ejemplos:

- uso de objetos compartidos

Comportamiento exigido

- Se quiere evitar auto-bloqueo debido a llamadas del mismo hilo para obtener la misma llave.
- Se quiere minimizar el trabajo adicional.

Posible solución

- Se aprovecha de las interfaces existentes en el lenguaje de programación para acceder a los componentes de una clase.
- Cada hilo accede solamente a métodos públicos mientras todavía no haya obtenido la llave.
- Dichos métodos públicos intentan obtener la llave cuanto antes y delegan después el trabajo a métodos privados (protegidos).
- Los métodos privados (o protegidos) asumen que se haya obtenido la llave.

Detalles de la implementación

- Los monitores de Java proporcionan directamente un mecanismo parecido al usuario, sin embargo, ciertas clases de Java (por ejemplo, tablas de dislocación (*hash tables*)) usan internamente este patrón por razones de eficiencia.
- Hay que tener cuidado de no corromper la interfaz, por ejemplo, con el uso de métodos amigos (`friend`) que tienen acceso directo a partes privadas de la clase.
- El patrón no evita bloqueo, solamente facilita una implementación más transparente.