Paradigma de programación dirigido por eventos

Omar Salvador Gómez Gómez

Diciembre 2007
versión 2.0

Tabla de Contenido

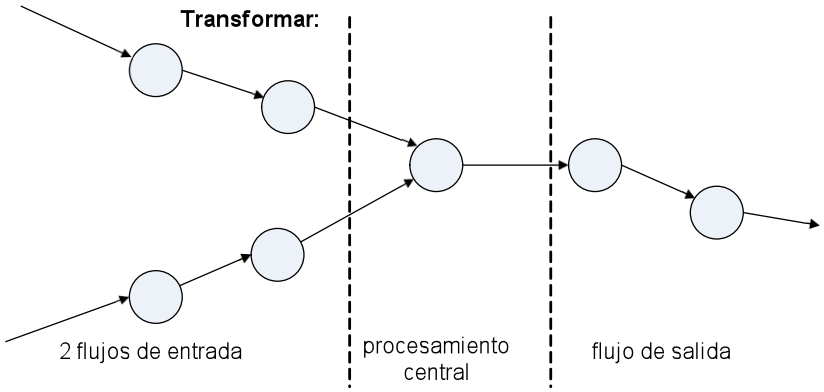
1. Introducción

En la actualidad una gran variedad de sistemas software hacen uso del paradigma dirigido por eventos, desde sistemas que utilizan interfaces gráficas de usuario (GUI) hasta sistemas complejos en tiempo real tales como sistemas controladores de vuelos. El objetivo del presente documento es dar a conocer al lector un panorama amplio sobre este paradigma. A continuación se describe la organización de éste, en la sección 2 se presentan los orígenes del paradigma dirigido por eventos, en la sección 3 se describe la estructura así como las características principales de este paradigma, en la sección 4 se presenta el estado del arte de en el que se incluye un breve resumen de los trabajos de investigación más relevantes que se han hecho en la actualidad, en la sección 5 se describen dos mecanismos usados en este paradigma que son, el manejo de hilos y el manejo de excepciones, finalmente en la sección 6 se presentan algunas conclusiones.

2. Orígenes del paradigma dirigido por eventos

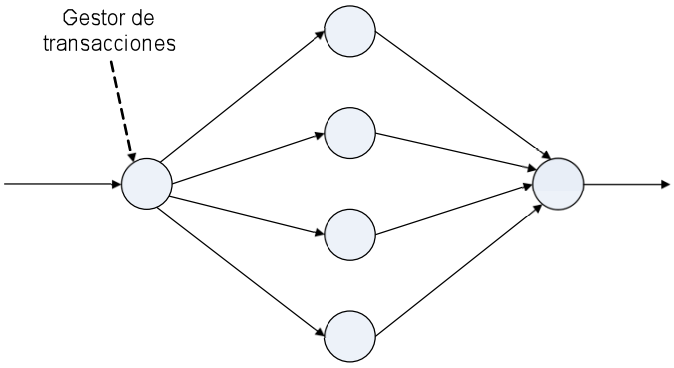
Los orígenes conceptuales de éste paradigma comienzan a finales de los años 70s con la llegada principalmente de un par de libros que tratan sobre métodos estructurados, *Análisis estructurado y especificación de sistemas* por De Marco[1] y, *Diseño estructurado* por Yourdon y Constantine[2]. A continuación se describe el origen del paradigma dirigido por eventos tomando como referencia los trabajos de los autores antes mencionados.

Una de las técnicas usada para el diseño estructurado fueron los diagramas de flujo de datos (DFD), éstos mostraban la estructura lógica de un sistema software. En un DFD, un registro de un archivo secuencial fue conceptualizado como un paquete de datos que viaja a través de un conducto continuo llamado *flujo de datos*. Los paquetes pasan a través de una secuencia de estaciones de trabajo llamados *procesos* en donde éstos son filtrados, utilizados, mejorados o transformados y después enviados a la siguiente estación de trabajo. A continuación se muestra un ejemplo de un diagrama de flujo de datos.



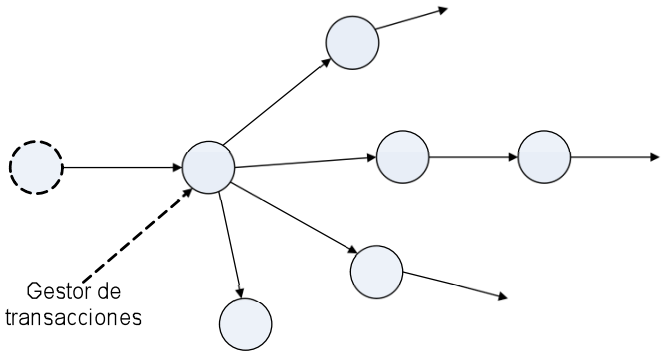
A este tipo de descripción de sistema se le llamó *análisis de transformación*.

De Marco, también describió brevemente un segundo tipo de análisis llamado *análisis de transacción* y definió el siguiente diagrama:



Las diferencias entre ambos diagramas son las siguientes, el diagrama de análisis de transformación utiliza procesos que transforman paquetes de datos, los procesos tienen claramente definidos sus flujos de entradas, procesamiento y flujos de salida. Una transformación es representada en un DFD en términos de una red linear. Sin embargo el diagrama de análisis de transacción se utiliza para modelar gestores de transacciones, en el que en un momento dado el flujo de paquetes puede representarse por un flujo de datos en paralelo.

De Marco le restó importancia al análisis transaccional, sin embargo este tema fue de gran interés para Yourdon y Constantine, quienes re-definen el diagrama propuesto por De Marco. En el diagrama propuesto por Yourdon y Constantine, un flujo de paquetes de datos pasa a través de una transformación (llamada así al gestor de transacciones) que divide el flujo de paquetes de entrada en varios sub-flujos discretos de salida. A continuación se muestra el diagrama:

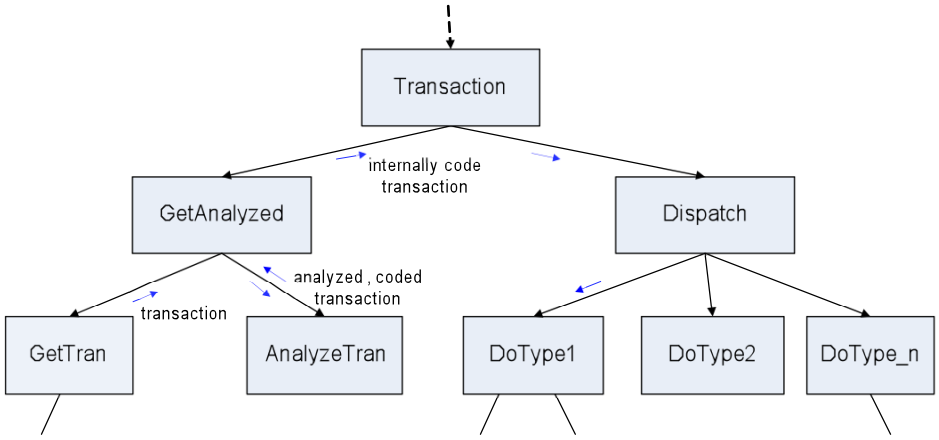


El diagrama propuesto por Yourdon y Constantine es el arquetipo del paradigma dirigido por eventos. Los autores definen que una transacción inicia cuando algún elemento de datos, control, señal, evento, o cambio de estado es enviado al proceso que conforma al gestor de transacciones.

El gestor de transacciones debe ser capaz de obtener y responder a las transacciones, analizar cada transacción para determinar su tipo, despachar de acuerdo al tipo de transacción y completar el proceso de cada transacción.

Un DFD muestra las funciones lógicas que un sistema debe efectuar. Sin embargo éste no muestra la información suficiente si se desea conocer el flujo de las funciones. Para tal propósito existe un tipo de diagrama llamado *estructura de cajas*. En este tipo de diagrama, las cajas representan módulos (funciones o subrutinas). Éstas son ordenadas jerárquicamente, los módulos que realizan llamadas se localizan en la parte superior mientras que los módulos llamados se ubican debajo de los módulos superiores.

A continuación se presenta un diagrama de análisis de transacción convertido en la representación de estructura de cajas.



En este diagrama la flecha punteada que viene de la parte superior representa el control de flujo que es pasado al gestor de transacciones. Las transacciones son recibidas por la función *GetTran*. Un vez obtenidas, la transacción es analizada para determinar su tipo (código/id de transacción) y después se envía al gestor de transacciones. De ahí estas son enviadas al módulo *Dispatch* el cual las reenvía al módulo responsable de manejar las transacciones de ese tipo.

En esta sección se ha hecho un breve recorrido por el tiempo, desde los primeros inicios de la programación hasta la concepción del paradigma dirigido por eventos. En la siguiente sección se describe la estructura y características de éste paradigma.

3. Estructura y características del paradigma dirigido por eventos

En esta sección se presentan de manera general dos clasificaciones de paradigmas de programación, la primera elaborada por Floyd[3] en el año de 1979 y la segunda realizada por Ambler[4] en el año de 1992, Este par de clasificaciones se presentan con la finalidad de dar a conocer al lector en cual de estas se encuentra ubicado el paradigma dirigido por eventos, en el resto de la sección se describe la estructura y características de este paradigma.

3.1. Clasificación de paradigmas según Floyd

Floyd describió tres categorías de paradigmas, aquellos que soportan técnicas de programación de bajo nivel, los que soportan métodos de diseño de algoritmos y aquellos que soportan enfoques de programación de alto nivel. De acuerdo a la clasificación de Floyd el paradigma dirigido por eventos está situado en la última categoría ya que este paradigma puede ser materializado en lenguajes de cuarta generación pudiendo hacer uso o no de la orientación a objetos así como de la utilización de múltiples hilos.

3.2. Clasificación de paradigmas de acuerdo a Ambler

Algunos años después, Ambler definió otra clasificación de paradigmas de acuerdo en como éstos solucionan algún problema determinado. La clasificación está formada de acuerdo a tres tipos de soluciones: operacional o procedimental, demostrativa y declarativa.

3.2.1. Solución operacional o procedimental

Describe las etapas necesarias para construir soluciones, señalando la forma de cómo obtenerlas. Se determina etapa a etapa el modo de construir la solución, describiendo cómo obtener un resultado a partir de un estado inicial.

En esta categoría se incluyen los lenguajes clásicos de la primera a tercera generación, incluyendo los orientados a objetos y los funcionales, que requieren desarrollar técnicas de depuración y verificación del funcionamiento del programa.

Los paradigmas procedimentales u operacionales, promueven secuencias computacionales, donde las variables que se relacionan con direcciones de memoria, pueden ser:

- Modificadas con efecto de lado. Esto es, que se actualizan iterando datos y direcciones de memoria de variables, registrando múltiples asignaciones hasta lograr el resultado. Sus tipos son paradigma imperativo y paradigma orientado a objetos.
- No Modificadas, sin efecto de lado. Son aquellos que van creando continuamente nuevos datos.

3.2.2. Solución demostrativa

Plantea la solución describiendo ejemplos y promueve que el sistema generalice la solución de tales ejemplos para otros casos.

Bajo este entorno denominado también "paradigma por ejemplo", la solución a un problema se logra aplicando la "programación por ejemplo" o demostrativa, donde el programador no requiere especificar con algún procedimiento el cómo lograr la solución, sino que resuelve programas similares, generalizando soluciones, ya sea por medio de simulación o por inferencias.

3.2.3. Solución declarativa

Describe qué se desea obtener, más no como obtenerla, indicando para ello, las características que debiera tener la solución, sin describir como procesarla.

En este caso el paradigma declarativo, aconseja construir soluciones acotando hechos, reglas, restricciones, ecuaciones, transformaciones y demás propiedades derivadas del conjunto de parámetros o valores que configuran la solución.

Con tal información, el sistema proporcionará un esquema que incluya la secuencia u orden de evaluación que genere una solución, sin describir las diferentes etapas a seguir para lograr tal solución.

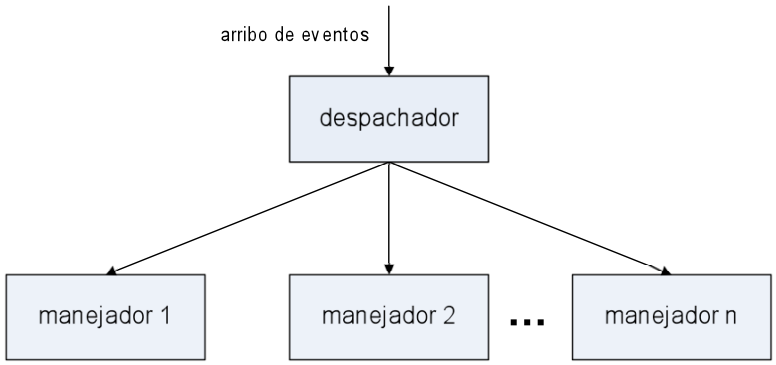
Esta solución usa variables para almacenar valores intermedios, pero no para actualizar estados de información y como especifican la solución sin indicar cómo construirla, anulan la necesidad de verificar que el valor calculado corresponde a la solución correcta.

De acuerdo a la anterior clasificación, el paradigma dirigido por eventos se sitúa dentro de la solución operacional o procedimental. Este paradigma puede hacer uso de los paradigmas Imperativo y Orientado a objetos para modelar la solución.

3.3. Patrones usados en el paradigma dirigido por eventos

El principal patrón estructural que constituye la esencia del paradigma dirigido por eventos es el llamado patrón Handler (manejador), el cual tiene su origen en los diagramas de análisis de transacciones definidos por De Marco, Yourdon y Constantine.

La mayoría de las materializaciones del paradigma dirigido por eventos hacen uso del patrón Handler. A continuación se muestra la figura que representa la estructura de éste.



Los elementos estructurales de este patrón son: un flujo de datos llamados eventos, un despachador de eventos y un conjunto de manejadores.

La función del despachador es tomar cada uno de los eventos que van llegando a el, analizar cada evento para determinar su tipo y finalmente enviar cada uno de éstos a los respectivos manejadores quienes realizan alguna función de acuerdo al tipo de evento que estos puedan procesar.

El despachador debe procesar un flujo de eventos entrantes, por lo que en la lógica de éste se debe incluir un bucle continuo para que el despachador pueda obtener cada evento que arribe, atenderlo y regresar al bucle para obtener y procesar el siguiente flujo de eventos entrante.

Algunos sistemas software tales como aquellos que controlan hardware pueden tratar efectivamente flujos de eventos continuos ó infinitos. Sin embargo en la mayoría de los sistemas dirigidos por eventos, los flujos de eventos son finitos. Éstos cuentan con un tipo de evento especial que representa el fin del flujo por ejemplo, la marca final de un fichero, el presionar la tecla *"escape"* o el presionar el botón izquierdo del Mouse sobre el botón "salir" de algún elemento de una GUI. En los anteriores casos en la lógica del despachador se debe incluir alguna capacidad de salir o romper el bucle continuo una vez que el final de un flujo de eventos ha sido detectado.

En algunas situaciones, el despachador puede determinar que este no cuenta con los manejadores adecuados para procesar algún tipo de evento. En estas situaciones, el despachador puede descartar el evento o disparar una excepción. Por ejemplo, algún programa que utilice interfaz grafica de usuario (GUI) estaría interesado en ciertos tipos de eventos como son, la presión en alguno de los botones del Mouse sobre algún elemento de la GUI, sin embargo a este programa no le seria de interés los eventos que representan el desplazamiento del Mouse de un punto a otro. En las aplicaciones GUI, los diferentes tipos de eventos que no cuentan con su respectivo manejador, son descartados. Para otro tipo de aplicaciones por ejemplo que no usen GUI, un evento no reconocido constituye un error en la entrada de flujo de eventos por lo que una acción apropiada es disparar una excepción.

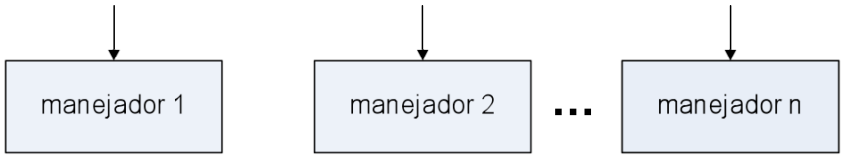
A continuación aparece una fracción en pseudo-código que representa a un despachador. Éste incluye las características antes descritas que son: un bucle continuo, una operación para salir, la determinación del tipo de evento así como la selección apropiada de un manejador para su procesamiento y finalmente el tratamiento de eventos que no cuentan con algún manejador para su tratamiento.

```
do forever:  # Bucle continuo

    obtener un evento del flujo de entrada

    if   event type == EndOfEventStream :
         quit # Sale del bucle

    if   event type == specific event type one :
         llamar al manejador apropiado de este evento

    elif event type == specific event type two :
         Llamar al manejador apropiado de este evento

    else:   # Si es algún tipo de evento desconocido
         ignore the event, or raise an exception
```

Existen otras variantes de este patrón, a continuación se describen más detalladamente.

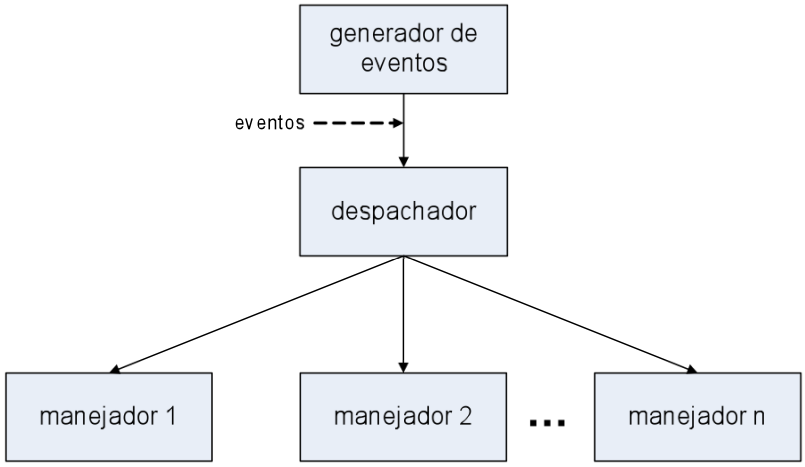
3.3.1. Patrón Handler sin cabeza

En este tipo de patrón, el despachador no está presente o no es visible. Sin la presencia explicita de éste lo único visible es una colección de manejadores de eventos, a continuación se muestra la estructura de esta variación.



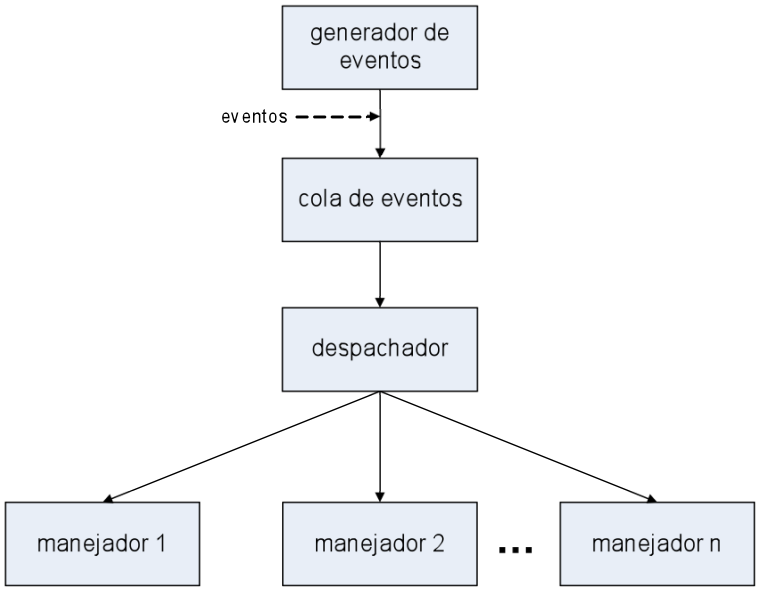
3.3.2. Patrón Handler extendido

En esta variante, el patrón incluye un componente generador de eventos el cual genera el flujo de eventos necesarios para que el despachador los procese. A continuación se muestra la estructura.



3.3.3. Patrón con manejo de colas de eventos

En algunos casos, el despachador y sus manejadores no son capaces de manejar los eventos tan rápido conforme estos van llegando. Para tales casos, la solución es colocar un almacén temporal llamado cola de eventos en donde se almacene el flujo de eventos conforme estos arriban. La cola de eventos se coloca entre el generador de eventos y el despachador. Los eventos son agregados al final de la cola tan rápido como estos arriban, el despachador remueve los eventos de la cola tan rápido como este es capaz de procesarlos. A continuación aparece la estructura de esta variante.



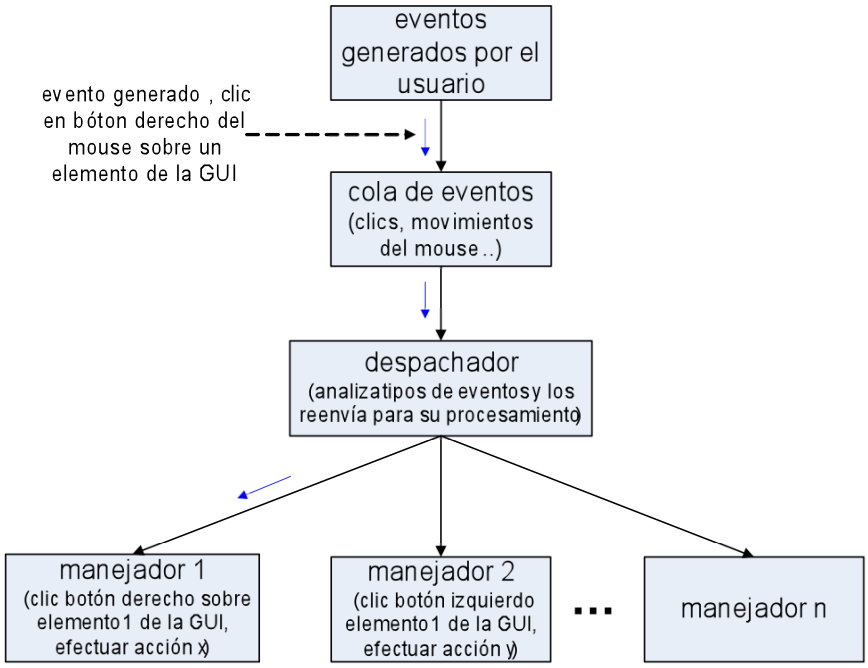
Las aplicaciones GUI típicamente incluyen una cola de eventos. Los eventos más significativos tales como la activación de los botones del Mouse pueden requerir algún tiempo determinado para ser manejados. Mientras esto sucede, otros eventos tales como movimientos del Mouse pueden acumularse en el almacén temporal. Cuando el despachador comienza a liberarse este puede descartar los eventos de movimiento del Mouse y removerlos del almacén temporal.

3.4. Usos del paradigma dirigido por eventos

Una vez descritos algunos de los patrones que conforman la estructura del paradigma dirigido por eventos, a continuación se explican algunos ejemplos de tipos de aplicaciones que en la actualidad hacen uso de este paradigma.

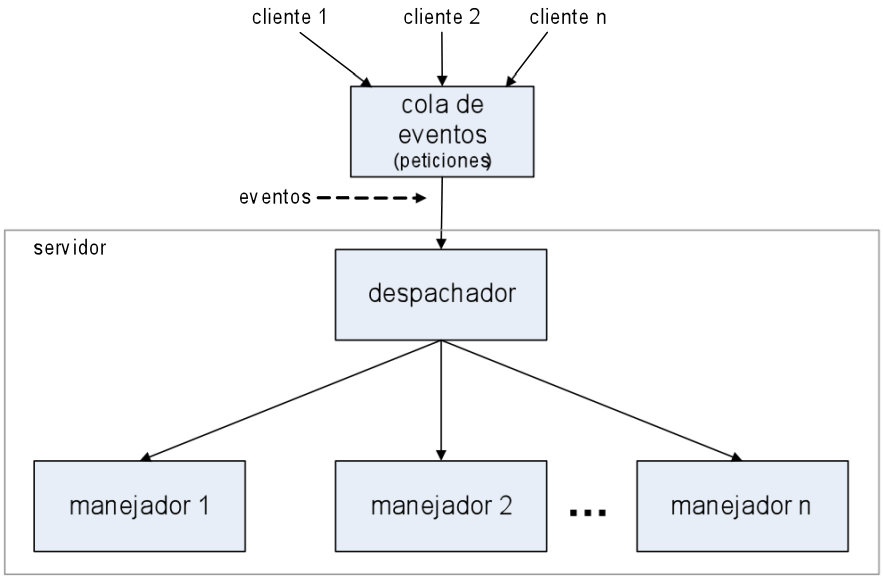
3.4.1. Sistemas GUI

En la actualidad la mayoría de las personas que utiliza un computador, hacen uso de sistemas que cuentan con interfaces gráficas de usuario (Graphic User Interface o GUI). La estructura de este tipo de sistemas implementa el patrón Handler. Cada elemento de una interfaz gráfica como lo son botones, cajas de selección, cajas de edición entre otros, pueden estar en modo de espera hasta la llegada de un evento como lo es el oprimir algún botón del Mouse sobre uno de los elementos de la GUI, una vez arribado el evento éstos realizan alguna acción. Los módulos que componen a este tipo de sistemas son los siguientes: un elemento que representa los eventos generados externamente por el usuario, un despachador responsable de analizar los eventos que llegan a éste y una serie de manejadores que efectúan alguna operación dependiendo del tipo de evento generado por el usuario. A continuación se muestra la estructura de un sistema GUI que implementa el patrón Handler.



3.4.2. Sistemas Cliente-Servidor

Un tipo de sistema que utiliza el patrón Handler son los llamados sistemas cliente-servidor. Este tipo de sistemas está compuesto por dos componentes principales uno o más servidores y uno o más clientes. El servidor puede ser un hardware o componente software cuya función es proporcionar un determinado tipo de servicio a los clientes quienes consumen el servicio. El trabajo de los servidores es esperar por las peticiones de los clientes, responder a las solicitudes de los clientes proporcionando el servicio y finalmente esperar por más peticiones. A continuación se muestra el patrón Handler en este tipo de sistemas.

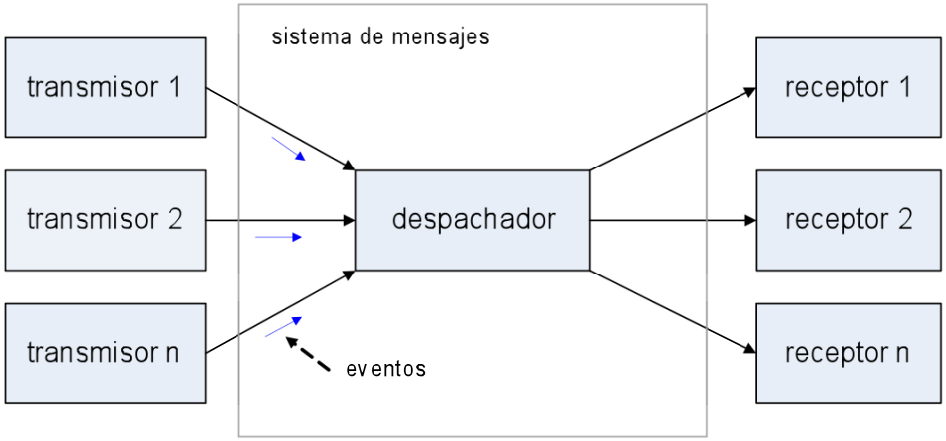


En el anterior diagrama, cada cliente solicita una serie de servicios específicos los cuales son ingresados en una cola de eventos, por su parte el servidor a través de un despachador va sacando de la cola de eventos los servicios solicitados y los delega a los diferentes manejadores de acuerdo al tipo de servicio.

3.4.3. Sistemas de mensajes

Otro tipo de variante del patrón Handler son los sistemas de mensajes. En este tipo de sistemas existen dos componentes, un generador de eventos llamado transmisor y un manejador de eventos llamado receptor. Por lo general los transmisores y receptores se encuentran en distintas ubicaciones físicas o en distintas plataformas. Este tipo de sistemas utiliza un modelo de conexión punto a punto ya que los transmisores conocen a que receptor enviar el mensaje.

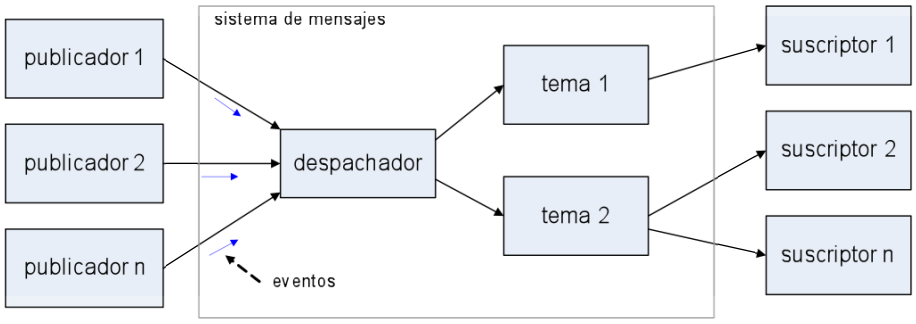
Un ejemplo de este son los sistemas de correo electrónico, un usuario envía un correo al servidor de correos (sistema de mensajes) el servidor obtiene la dirección del destinatario y lo envía al receptor. A continuación aparece la estructura de este tipo de sistemas.



Existen otros sistemas de mensajes más sofisticados llamados sistemas de mensajes empresariales, los cuales utilizan un *middleware* orientado a mensajes (*message oriented middleware* o MOM). Una de las características de este tipo de sistema es que pueden contener diferentes programas o componentes hechos en distintos lenguajes y ubicados en diversos sitios, estos pueden operar en diferentes plataformas y tener la capacidad de comunicarse entre ellos a través de envío de mensajes. Este tipo de sistemas utiliza un modelo de conexión conocido como publicador-suscriptor.

En este modelo, los receptores son conocidos como suscriptores ya que estos registran una serie de eventos que les interesa procesar de acuerdo a una clasificación de temas. Por su parte, los transmisores conocidos como publicadores envían una serie de eventos a los temas registrados. Cuando un tema recibe un mensaje, este verifica a los

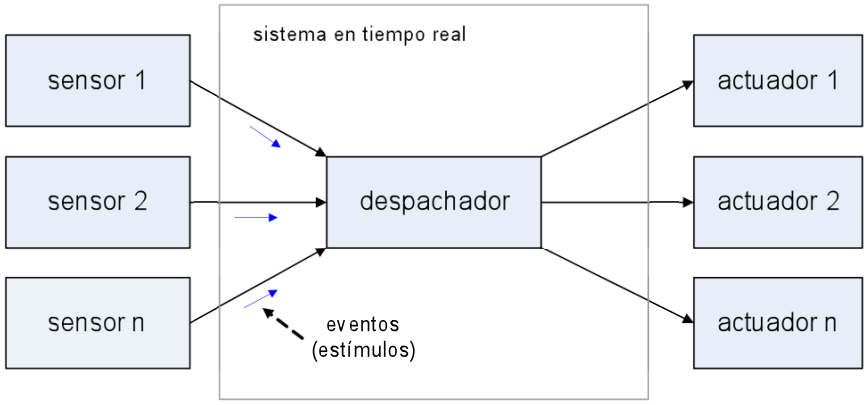
suscriptores que se encuentran registrados en este tema y reenvía el mensaje a los suscriptores seleccionados para su procesamiento. A continuación se ilustra la estructura del modelo publicador-suscriptor.



3.4.4. Sistemas en tiempo real

Otro tipo de sistemas que hacen uso del patrón handler, son los denominados en tiempo real, básicamente este tipo de sistema se encuentra embebido en diferentes tipos de hardware que controlan desde máquinas domésticas hasta plantas enteras de manufactura.

Un sistema en tiempo real debe reaccionar a eventos generados por el hardware y emitir señales de control como respuesta a dichos eventos. Estos sistemas se componen de los siguientes elementos: una serie de sensores responsables de generar eventos llamados estímulos, un componente responsable por el control de los eventos (despachador) y una serie de elementos llamados actuadores (manejadores) responsables de dar una respuesta al evento recibido. A continuación se ilustra la estructura de este tipo de sistemas.



4. Estado del arte

En esta sección se presenta una breve recopilación sobre los trabajos actuales más significativos con respecto al paradigma dirigido por eventos. Éstos incluyen investigaciones llevadas a cabo en el desarrollo de *frameworks*, metodologías y lenguajes de programación.

El grupo Gartner [5] pronostica que los próximos sistemas software serán más orientados a eventos. Este pronóstico se basa en cinco fuerzas que se encuentran detrás de esta tecnología, a continuación se describen.

1. Estrategias de negocio que demandan diseños dirigidos por eventos. Existen enormes beneficios tanto financieros como estratégicos para implementar procesos de negocio dirigidos a eventos, ya que estos se adaptan de manera inherente en muchos aspectos del mundo real por ejemplo, los negocios de hoy en día demandan aspectos tales como cambios en los procedimientos de operación, en flujos de trabajo y en las relaciones con los clientes y proveedores.

2. SOA está ayudando a educar a los desarrolladores con respecto a la computación distribuida. La amplia escala de migraciones a arquitecturas orientadas a servicio (SOA) está ayudando a preparar el terreno para una mayor adopción de sistemas dirigidos por eventos, ya que SOA y la computación dirigida por eventos están basadas en componentes de negocio distribuidos. Muchas de las características como modularidad, encapsulación y documentación de interfases que se aplican a los componentes SOA también se aplican a los componentes dirigidos por eventos.

3. Existencia en el mercado de productos que soportan esta tecnología. Los sistemas *midleware* tales como aquellos orientados a mensajes y servicios web se encuentran disponibles por una variedad de proveedores. Los sistemas dirigidos por eventos más sofisticados que permiten transformaciones, direccionamiento basado en contenido y administración de procesos de negocio (BPM) ya están listos para ser integrados por agentes intermediarios (*brokers*) y motores BPM.

4. Los estándares serán la clave. Los estándares en la industria de sistemas dirigidos por eventos están incompletos, sin embargo conforme estos maduren se harán más populares los conceptos de eventos y estos traerán notables consistencias entre los diferentes sistemas software que usen esta tecnología.

5. Las mejoras en el hardware y en telecomunicaciones serán de ayuda. Los continuos mejoramientos en hardware, así como el incremento de un mayor desempeño en las telecomunicaciones está permitiendo cada vez más la incorporación de este tipo de sistemas en el sector empresarial.

Fiege, Mühl y Gärtner [6], presentan una metodología para construir sistemas dirigidos por eventos en la que hacen uso de dos métodos estructurados que son, el uso de

ámbitos[1] y mapeo de eventos. En este trabajo se presenta un ejemplo de diseño modular y la implementación de un sistema dirigido por eventos utilizando dicha metodología. De acuerdo a los autores, el método propuesto facilita la coordinación entre componentes en un sistema dirigido por eventos.

Petitpierre y Eliëns [7] proponen el uso de Objetos Activos para el desarrollo de sistemas dirigidos por eventos. Uno de los problemas que tiene esta clase de sistemas es el no-determinismo inherente, por lo que es difícil desarrollar y mantener este tipo de sistemas. Por ejemplo, en una aplicación GUI un número indefinido de manejadores pueden registrar un conjunto de eventos para su procesamiento, sin embargo es difícil visualizar el control de flujo de estos eventos, a su vez es difícil determinar el orden en que éstos se deben procesar.

La solución que proponen los autores a este problema es el uso de llamadas síncronas, las cuales pueden ser introducidas en cualquier lenguaje de programación orientado a objetos. Estas llamadas son ejecutadas entre varios objetos activos[2].

Desde una perspectiva de la ingeniería esta propuesta promueve un enfoque más modular a los sistemas dirigidos por eventos así como una mejor visualización en el flujo de control entre los componentes responsables de recibir los eventos y la aplicación. Este enfoque permite una forma más directa de interceptar eventos, por lo que se obtiene una simplificación significativa de código. El control de flujo llega a ser mas fácil de comprender por el uso de comunicaciones síncronas entre los objetos. El enfoque propuesto hace uso de JCP, una extensión basada en CSP (*Concurrent Sequential Processes*).

Fenkam, Gall y Jazayeri [8], proponen una metodología y un *framework* para el desarrollo de sistemas dirigidos por eventos llamado LECAP (*Logic of Event Consumption And Publication*). Uno de los problemas actuales que se tiene en el desarrollo de sistemas dirigidos por eventos es que estos se elaboran de manera ad-hoc debido a la carencia de metodologías, por consecuencia no es posible realizar razonamientos formales que soporten la corrección[3] de éste. La metodología y framework propuestos ayudan a resolver este problema, a través de un paradigma de composición y refinamiento que permite especificar y verificar la corrección de los sistemas dirigidos por eventos.

Aunque no se espera que los desarrolladores lleven a cabo rigurosas técnicas formales que permitan razonar con respecto a la corrección de los sistemas que desarrollan durante las fases de diseño e implementación, el uso de este tipo de técnicas ayuda a

[1] Un ámbito envuelve un conjunto de productores y consumidores (publicadores y suscriptores) y delimita su visibilidad de los eventos publicados. Los ámbitos pueden re-publicar eventos internos y remitir eventos externos a sus miembros, y de este modo un ámbito puede ser visto como un productor y consumidor. Éste puede recursivamente ser un miembro de otro ámbito, ofreciendo de esta manera un mecanismo poderos de estructuración.

[2] Un objeto activo se caracteriza por tener un hilo propio de ejecución, por ejemplo en el lenguaje Java un objeto activo es aquel que cuenta con un método propio run() que efectúa alguna acción en particular.

[3] En la teoría de las ciencias computacionales, la corrección de un algoritmo se afirma cuando se dice que éste es correcto con respecto a su especificación.

obtener intuiciones que pueden ser invaluables para los desarrolladores, más aún, estas técnicas pueden ser aplicadas sin tanto rigor.

El *framework* LECAP se basa en obtener la composición de un sistema dirigido por eventos a un nivel abstracto en donde las propiedades de éste puedan ser verificadas más fácilmente que a nivel concreto. Los autores proponen dos formas para especificar la abstracción de estos sistemas, una es especificar su estructura y la otra su comportamiento.

En la especificación estructural, uno puede especificar el arribo de eventos de forma independiente. Esto es útil para especificar componentes, ya que el contexto en el cual el componente será desarrollado no es necesario que sea conocido. La especificación de comportamiento se extrae de la especificación estructural y es solamente usada para la verificación de propiedades.

La metodología propuesta consta de cuatro pasos, y a continuación se describen:

1. Diseñar la arquitectura del sistema. Identificar sus componentes.
2. Desarrollar formalmente la especificación estructural de los componentes y verificar algunas de sus propiedades.
3. Elaborar de manera formal la especificación de comportamiento de todo el sistema.
4. Efectuar un refinamiento de la especificación.

Esta metodología es una combinación de los enfoques *bottom-up* y *top-down*. En el enfoque *bottom-up* se construyen las especificaciones de los componentes y se verifican las propiedades del sistema, mientras que en el enfoque *top-down* los componentes previamente especificados pueden ser construidos. El uso de estos dos enfoques es útil para el desarrollo de sistemas basados en componentes.

El *framework* propuesto incluye cuatro elementos:

1. Una definición formal de un lenguaje que soporta el desarrollo de sistemas dirigidos por eventos, este lenguaje es llamado lenguaje de programación LECAP.
2. El modelado formal de un sistema dirigido por eventos.
3. Un método para la especificación de este tipo de sistemas que incluye dos técnicas de especificación: especificaciones estructurales y de comportamiento.
4. Un conjunto de reglas para el desarrollo de sistemas en el enfoque *top-down*.

Sin embargo, el trabajo antes descrito se encuentra en proceso de maduración ya que ha sido validado únicamente en unos cuantos casos de estudio.

El trabajo de Chen [9] se basa en el uso de un *framework* para verificar la corrección de aplicaciones GUI desarrolladas en Java con las librerías swing o awt, esta verificación se lleva a cabo a través del uso de modelos formales.

Los programas Java que hacen uso de Swing o AWT pueden experimentar demasiados fallos debido al no determinismo implícito causado por el complejo manejo de eventos que es efectuado a través de múltiples hilos.

Chen, propone un framework para modelar el mecanismo de manejo de eventos en una aplicación GUI hecha en Java utilizando sistemas de transición de etiquetas[4]. Las semánticas operacionales propuestas, dan una definición precisa del comportamiento del sistema con respecto a los eventos GUI, por lo que ayudan al desarrollador a obtener un mejor entendimiento del comportamiento del sistema y se benefician de evitar una codificación incorrecta.

Este *framework* puede ser utilizado de dos formas, una para proporcionar una base formal para un mejor entendimiento y correcto uso de los mecanismos en los eventos de las aplicaciones Java GUI y la segunda, para razonar formalmente con respecto a la corrección de una aplicación Java GUI contra ciertas propiedades que debido al no determinismo involucrado, pudieran ser difíciles de detectar por las técnicas de pruebas existentes.

El trabajo de Fischer, Majumdar y Millstein [10] consiste en la elaboración de un modelo de programación llamado *Tasks* el cual permite desarrollar sistemas dirigidos por eventos a través de la implementación de un conjunto de extensiones para el lenguaje de programación Java que dan soportan a este modelo. Como se ha mencionado en los trabajos anteriores, los sistemas hechos bajo el paradigma dirigido por eventos complican el mantenimiento y el entendimiento de éstos.

El modelo propuesto es un variante de la cooperación entre múltiples hilos y permite a cada flujo de control lógico ser modularizado de una manera tradicional, incluyendo el uso de mecanismos de control estándar tales como procedimientos y excepciones. Los autores introducen el concepto de tareas, en donde una tarea es como un hilo la cual encapsula una unidad de trabajo independiente.

El flujo de control lógico de cada unidad de trabajo es preservado, y las estructuras del programa comunes tales como procedimientos y excepciones pueden ser usadas de una manera natural. Sin embargo a diferencia de los hilos, las tareas pueden ser implementadas de manera automática por el compilador conforme al paradigma dirigido por eventos. *TaskJava* es el nombre que reciben las extensiones desarrolladas que dan soporte a este modelo de programación.

Este trabajo se encuentra en la fase de maduración, ya que solamente ha sido probado en algunos casos de estudio. De acuerdo a los autores, es el primer paso hacia alcanzar la meta de escribir códigos robustos y confiables para este tipo de sistemas.

En esta sección se han presentado las contribuciones más relevantes realizadas en la actualidad con respecto al paradigma dirigido por eventos, éstas se centran básicamente en la verificación de un conjunto de propiedades sobre este tipo de sistemas.

---

[4] Un sistema de transición de etiquetas es una terna (estados, etiquetas,→) en donde: los estados son un conjunto posible de estados de un programa, las etiquetas son un conjunto de etiquetas que muestran la información acerca de los cambios de estados y → es la relación de transición que describe la evolución del sistema con respecto a sus estados.

5. Modelo de programación

En esta sección se describen dos mecanismos que están presentes en el paradigma dirigido por eventos que son, el uso de hilos [11] y el manejo de excepciones [12]. Posteriormente se describe un ejemplo de un programa desarrollado bajo éste paradigma. Cabe mencionar que tanto los mecanismos a ser descritos como el ejemplo están basados en el lenguaje de programación Java.

5.1. Uso de Hilos en el paradigma dirigido por eventos

Uno de los mecanismos que es usado en este paradigma es la utilización de hilos, por ejemplo, los manejadores de eventos son implementados como hilos, ya que éstos continuamente deben estar esperando por el arribo de eventos para su procesamiento, a su vez varios manejadores pueden procesar más de un evento en un mismo periodo de tiempo por lo que el uso de múltiples hilos es común en este paradigma.

Conceptualmente, un Hilo es un flujo de control dentro de un programa. Un hilo es similar a la familiar noción de proceso, excepto que varios hilos dentro de la misma aplicación comparten el mismo estado. En particular, éstos corren dentro del mismo espacio de direcciones. El compartir el mismo espacio de direcciones significa que los hilos comparten instancias de variables pero no las variables locales.

En la actualidad varios lenguajes de programación soportan el manejo de hilos, lo que permite el desarrollo de programas bajo el paradigma dirigido por eventos. En el lenguaje de programación Java, existen dos maneras de crear hilos, extendiendo de la clase *Thread* o implementando la interfaz *Runnable*.

La primera forma de crear un hilo es simplemente extender alguna clase de la clase *Thread*. La clase *Thread* está definida en el paquete *java.lang* por lo que se requiere sea importado en su respectiva definición.

```
import java.lang.*;
public class RegisterStudentComponent extends Thread
{
   public void run()
   {
    ....
   }
}
```

El ejemplo anterior crea una nueva clase llamada *RegisterStudentComponent* que extiende de la clase *Thread* y ésta sobrescribe al método *run()* por su propia implementación.

La misma clase puede ser creada implementando la interfaz *Runnable*. En esta clase el método abstracto *run()* es definido en la interfaz *Runnable*.

```
import java.lang.*;
public class RegisterStudentComponent implements Runnable
{
        Thread T;
         public void run()
        {
        ....
        }
}
```

La diferencia entre lo dos fragmentos de código anteriores es que la clase que implementa la interfaz *Runnable* tiene mayor flexibilidad ya que puede extender funcionalidad extra de otras clases.

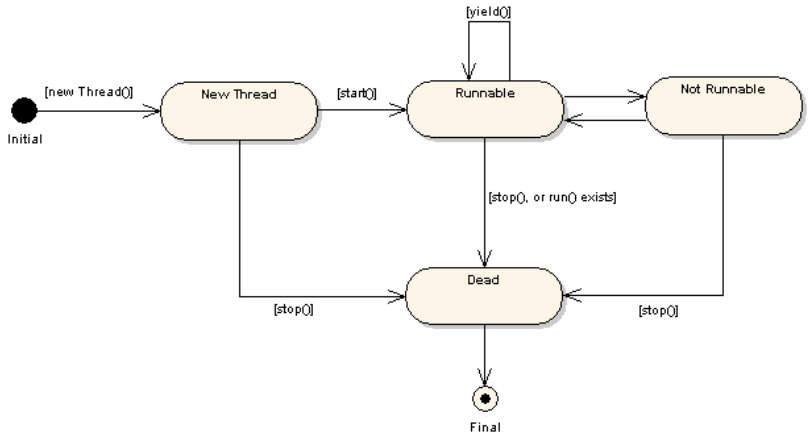
Una interfaz solo proporciona un diseño de cómo las clases deben ser implementadas, a continuación se muestra la definición de la interfaz *Runnable*.

```
package java.lang;
public interface Runnable
{
    public abstract void run();
}
```

En este caso, la interfaz *Runnable* obliga a que solamente exista la definición del método *run()* por lo tanto, la mayor parte del trabajo es hecho por la clase *Thread*.

5.1.1. Ciclo de vida de un hilo

Un hilo puede encontrarse en varios estados durante su ciclo de vida, a continuación se muestra una figura que indica los estados y transiciones de un hilo, cabe señalar que en el estado de *Not Runnable* se están considerando los estados de *Waiting*, *Sleeping* y *Blocked*.



5.1.2. Estado Hilo nuevo (new Thread)

La siguiente instrucción crea un nuevo hilo sin embargo éste no inicia, por lo que el hilo se mantienen en el estado de *New Thread*.

```
Thread registerStudent =  new RegisterStudentComponent();
```

Cuando un hilo se encuentra en el estado de *New Thread*, este es simplemente un objeto *Thread* vacío. Los recursos de sistema aun no son asignados a éste, por lo que cuando un hilo se encuentra en este estado las dos posibles transiciones que se pueden efectuar son, iniciar o parar el hilo.

5.1.3. Estado Runnnable

En las siguientes líneas de código, el método *start()* crea los recursos del sistema necesarios para ejecutar el hilo, planifica el hilo para su ejecución y llama al método *run()* del hilo.

```
Thread registerStudent =  new RegisterStudentComponent();
registerStudent.start();
```

En este punto el hilo se encuentra en el estado *Runnable* o en ejecución. Por lo que las instrucciones definidas dentro del método *run()* son ejecutadas de manera secuencial.

5.1.4. Estado Not Runnnable

Un hilo puede entrar en el estado *Not Runnable* cuando uno de estos cuatro eventos ocurre:

- Se invoca al método *sleep()*
- Se invoca al método *suspend()*
- El hilo utiliza su método *wait()* para esperar sobre una condición de alguna variable
- El hilo se encuentra bloqueado por algún mecanismo de I/O

Por ejemplo en el siguiente fragmento de código se pone al hilo a dormir por un lapso de 10 segundos.

```
Thread registerStudent =  new RegisterStudentComponent();
registerStudent.start();
registerStudent.sleep(10000);
```

Durante 10 segundos el hilo *registerStudent* se queda dormido, incluso si el procesador quedara disponible el hilo no se ejecuta. Después de 10 segundos el hilo vuelve a el estado *Runnable* y, si el procesador está disponible efectúa la ejecución de éste.

5.1.5. Estado Dead

Un hilo puede morir o ser destruido de dos maneras, por causas naturales o porque se paró la ejecución de éste. Por ejemplo un hilo muere de manera natural cuando no hay algún bucle infinito dentro del método *run()*. En el siguiente ejemplo se hace una iteración 100 veces y termina la ejecución del hilo por lo que este muere de manera natural.

```
public void run() {
     int i = 0;
     while (i < 100) {
         i++;
         System.out.println("i = " + i);
     }
 }
```

Un hilo también puede morir en cualquier momento si se invoca al método *stop()*. En el siguiente fragmento de código se crea un hilo se pone a dormir por 10 segundos y finalmente se destruye con la invocación del método *stop()*.

```
Thread registerStudent =  new RegisterStudentComponent();
registerStudent.start();
registerStudent.sleep(10000);
registerStudent.stop();
```

El método *stop()* lanza un objeto del tipo *ThreadDeath* al hilo para matarlo. Así, cuando un hilo es muerto de esta forma este muere de manera asíncrona. El hilo morirá cuando reciba la excepción *ThreadDeath*.

El método stop() causa una terminación repentina del método *run()* del hilo. Si el método *run()* realiza cálculos críticos, la llamada a stop() puede dejar el programa en un estado inconsistente.

Finalmente existe un método para determinar si un hilo se encuentra iniciado o no, el método es *isAlive()* y regresa *true* si el hilo se encuentra en el estado de "*Runnable*" o "*Not Runnable*", si el hilo se encuentra en el estado de "*New Thread*" o "*Dead*" este método regresa *false* .

5.2. Manejo de excepciones en el paradigma dirigido por eventos

El segundo mecanismo comúnmente utilizado en este paradigma es el manejo de excepciones. A través de este mecanismo es posible dar un tratamiento especial a un programa cuando este presenta condiciones anormales o condiciones de excepción.

Una condición de excepción es un problema que previene la continuación de un programa. En este tipo de condición no es posible continuar con la ejecución del código porque no se tiene la información necesaria para resolver el problema en el contexto actual. Lo único que se puede hacer en esta situación es saltar del contexto actual y delegar el problema a un contexto superior. Esto es lo que sucede cuando se lanza una excepción.

En el lenguaje de programación Java las excepciones son objetos. Cuando se lanza una excepción se lanza un objeto. Los únicos objetos que se pueden lanzar como excepciones son aquellos que descienden de la clase *Throwable*. *Throwable* tiene dos subclases directas, *Exception* y *Error*. Las Excepciones, que son clases de la familia *Exception* son lanzadas cuando sucede alguna condición anormal en el flujo de algún programa, estas pueden ser tratadas de manera especial. Los errores que son miembros de la familia *Error* usualmente son lanzados cuando suceden problemas mas serios tales como desbordamientos de memoria y estos no pueden ser fácilmente tratados. Los errores comúnmente son lanzados por la maquina virtual.

Para lanzar alguna excepción dentro del lenguaje de programación Java, simplemente se usa la cláusula *throw* con un objeto de referencia, por ejemplo:

```
throw new FechaHoraInicioException();
```

En el siguiente fragmento de código, dentro del método *inciarSubasta()* se lanza una excepción del tipo *EntradaNoValidaException* si la condición es verdadera.

```
public void iniciarSubasta(Subasta subasta) throws
  EntradaNoValidaException
{
  if (subasta.getIdSubasta() == null )
      throw new EntradaNoValidaException
                  ("Id de subasta no debe estar en blanco");
  else
       subastaFacade.create(subasta);
}
```

Así como es posible lanzar excepciones, también es posible atrapar éstas. Para atrapar alguna excepción se debe escribir un bloque *try* con una o más cláusulas *catch*. Cada cláusula *catch* especifica un tipo de excepción que es preparado para ser tratado. El código que puede ocasionar alguna condición anormal del programa debe ir dentro de un bloque *try-catch*.

```
try
{
   Subasta subasta = new Subasta();
   SubastadorMgrRemote subastador = this.lookupSubastadorFacade();
   subastador.iniciarSubasta(subasta);
}
catch (EntradaNoValidaException ex) {
    ex.printStackTrace();
}
catch (FechaHoraNoValidaException ex) {
    ex.printStackTrace();
}
```

Como se muestra en el fragmento de código anterior, se le da tratamiento especial a dos tipos de excepciones que pueden presentarse si se genera algún flujo anormal en el código se encuentra dentro de las cláusulas *try-catch*.

En el paradigma dirigido por eventos se puede hacer uso del mecanismo de excepciones para darle tratamiento a los diferentes tipos de condiciones anormales que se pudieran presentar durante la comunicación entre los diferentes manejadores de eventos que forman parte de un sistema software.

6. Conclusiones

En el presente documento se ha dado a conocer el paradigma dirigido por eventos, desde sus orígenes hasta su uso en la actualidad. Cabe señalar que otros paradigmas se encuentran en un estado de mayor madurez. Por ejemplo, el paradigma orientado a objetos cuenta con una gran variedad de métodos y lenguajes de programación que soportan a este paradigma. Sin embargo, aunque en la actualidad el uso de sistemas software dirigidos a eventos es cada vez mayor, aún no existe alguna metodología lo suficientemente madura que sea capaz de dar soporte a este paradigma, así como tampoco existen leguajes de programación especializados en implementar este paradigma. Ha habido pequeños avances significativos en el desarrollo de metodologías y lenguajes de programación, sin embargo estas metodologías han sido validadas únicamente con experimentos controlados.

7. Referencias

[1] Tom DeMarco. *Structured Analysis and System Specification.* , Upper Saddle River, NJ, USA, Prentice Hall PTR, 1979.

[2] Edward Yourdon & Larry L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design.* , Upper Saddle River, NJ, USA, Prentice-Hall, Inc., 1979.

[3] Robert W. Floyd. *The paradigms of programming.* Commun. ACM, New York, NY, USA, Volume 22, Number 8, Pages 455-460, ACM, 1979.

[4] Allen L. Ambler; Margaret M. Burnett & Betsy A. Zimmerman. *Operational versus definitional: a perspective on programming paradigms.* Computer, Los Alamitos, CA, USA, Volume 25, Number 9, Pages 28-43, IEEE Computer Society Press, 1992.
[5] Roy W. Schulte. *The Growing Role of Events in Enterprise Applications.* Gartner research publication, Pages 5, 2003.
[6] Ludger Fiege; Gero Mühl & Felix C. Gärtner. *A modular approach to build structured event-based systems.* SAC '02: Proceedings of the 2002 ACM symposium on Applied computing, New York, NY, USA, Pages 385-392, ACM, 2002.
[7] C. Petitpierre & A. Eliëns. *Active Objects Provide Robust Event--driven Applications.* SERP'02, Las Vegas, June, Pages 253-259, 2002.
[8] Pascal Fenkam; Harald Gall & Mehdi Jazayeri. *A Systematic Approach to the Development of Event Based Applications.* srds, Los Alamitos, CA, USA, Volume 00, Pages 199, IEEE Computer Society, 2003.
[9] Jessica Chen. *Formal Modelling of Java GUI Event Handling.* ICFEM '02: Proceedings of the 4th International Conference on Formal Engineering Methods, London, UK, Pages 359-370, Springer-Verlag, 2002.
[10]Jeffrey Fischer; Rupak Majumdar & Todd Millstein. *Tasks: language support for event-driven programming.* PEPM '07: Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, New York, NY, USA, Pages 134-143, ACM, 2007.
[11] Scott Oaks & Henry Wong. *Java Threads, 2nd edition*. , O'Reilly, 1999.
[12] Bruce Eckel. *Thinking in Java, 2nd edition*. , Prentice Hall, 2000.