Programación basada en/dirigida por eventos “Event-Based Programming”: Conceptos (o Event-driven Programming)

ELO329: Diseño y Programación Orientados a Objetos

Introducción

- Hasta ahora nuestros programas ejecutan instrucciones siguiendo el orden de llamados definido en método main (*).
- Ésta no es la forma como operan los programas con interfaces gráficas. En éstos el programa debe responder cada vez a una solicitud del usuario en alguna de las partes de la interfaz gráfica.
- ¿Cómo monitorear dónde el usuario hará el siguiente click? Para esto ayuda la programación conducida por eventos.

(*) Una excepción a esto fue la clase Timer cuando estudiamos clases anidadas.

Programación basada en eventos: Otro paradigma de programación

- La programación basada en eventos (o programación dirigida por eventos) es aquella donde el flujo del programa está determinado por eventos; por ejemplo, salidas de un sensor, eventos de usuario (mouse, teclado), mensajes desde otros programas (locales o remotos), etc.
- También puede ser definida como una técnica para estructurar aplicaciones en donde la aplicación tiene un loop principal con las siguientes dos secciones:
  - Selección o detección de evento
  - Manejo o reacción frente al evento

```
Inicialización;
Forever do {
        Detecte la llegada de un evento;
        Ejecute acción definida para ese evento;
}
```

Otro paradigma de programación (cont.)

- En sistemas embebidos (micro-procesadores / microcontroladores, etc) y para responder a eventos de hardware, este modelo se implementa usando interrupciones en lugar de correr un loop infinito.
- Programas basados en eventos pueden ser escritos en cualquier lenguaje pero esta tarea se facilita en los lenguajes que proveen abstracciones de alto nivel para ello.
- En Java el loop infinito lo proporciona el ambiente gráfico de Java. El programador solo debe registrar el código a ejecutar ante la ocurrencia de cada evento de interés.

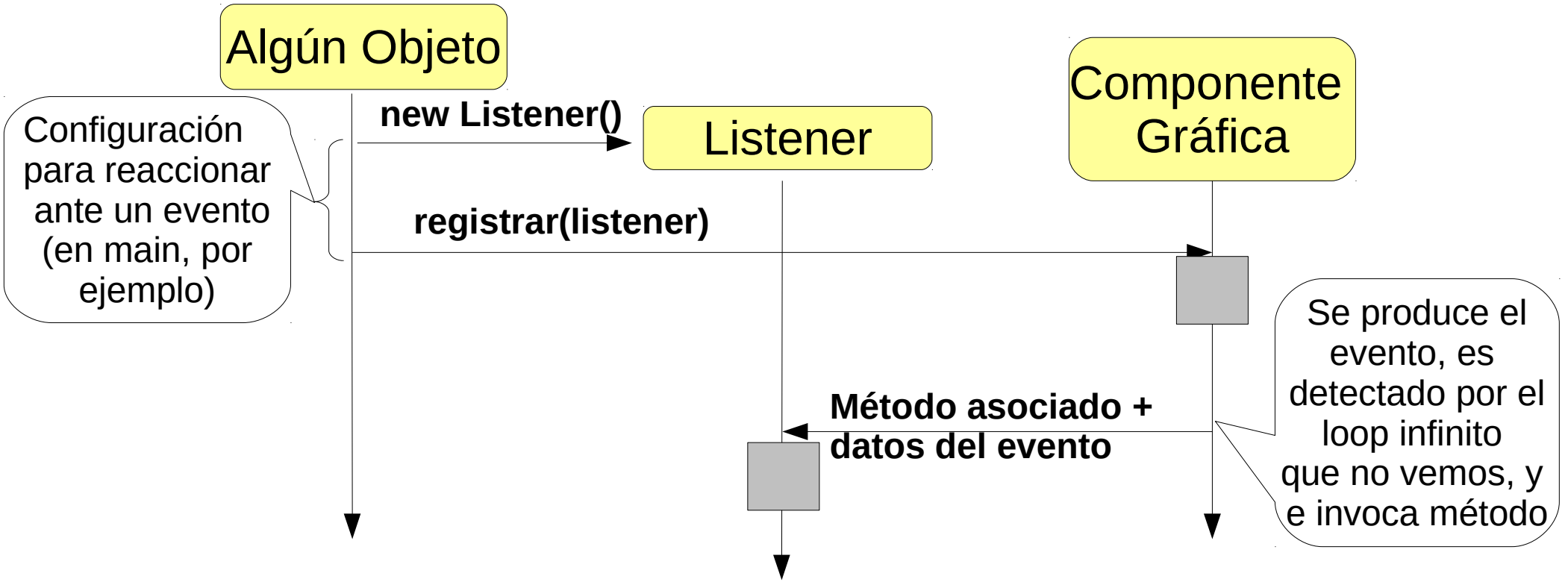
Uso de programación de eventos en GUIs

- Este modelo tiene gran aplicación en la programación de interfaces gráficas de usuarios.
- Hasta ahora, los programas de consolas típicos siguen un flujo secuencial en que típicamente se tienen ciclos:
  - entrada->procesamiento->salida
- Cuando programamos una Interfaz Gráfica de Usuario (GUI: Graphics User Interface) debemos tomar en cuenta la variedad de posibles interacciones con el usuario.
- En lugar de un único flujo de entrada de datos por consola, las GUIs permiten muchas más acciones del usuario.
  - Por ejemplo: es posible presionar botones gráficos, escribir texto en un campo de texto, o mover algún scrollbar.
  - ¿Cómo podemos estar atentos a tantas cosas al mismo tiempo?

Modelo

- Una forma de manejar todo tipo de posibles interacciones de usuarios es el uso de interrupciones; por ejemplo, cuando suena el timbre o teléfono en casa.
- De esta manera la CPU no pierde tiempo “mirando” los posibles eventos de usuarios, sino simplemente responde al evento y reanuda su procesamiento normal (otras tareas).
- Lenguajes como Java nos permiten definir y manejar interrupciones o eventos por software.
- La API de Java permite crear clases de objetos, llamados listeners, que responden a eventos causados por la GUI.
- La API de Java tiene interfaces (listeners) que deben ser implementadas por las clases que atenderán (manejará) los eventos de interés.
- Los métodos de la interfaz (“equivalen a las rutinas de servicio de interrupción”) son llamados cuando un evento específico ocurre.

Pasos para programar respuestas a Eventos



- Un **listener** es una instancia de una clase definida por el usuario. Esta clase implementa los métodos de una Interfaz (estos métodos equivalen a las rutinas para atender los eventos de interés).
- Las componentes gráficas de Java definen el métodos asociados a eventos y los datos pasados al invocarlo.

Ejemplo

- Veamos el caso de una ventana de nivel superior en Java (son aquellas que se pueden mover libremente en la pantalla - desktop) CloseableFrame.java

Clase que “activa” el loop infinito

```
import java.awt.event.*;
import javax.swing.*;

class CloseableFrame extends JFrame {
    public CloseableFrame() {
        setTitle("My Closeable Frame");
        setSize( 300, 200);
       // Creamos y registramos el objeto que se hará
       // cargo de atender eventos de la ventana
        MyWindowListener listener = new MyWindowListener();
        addWindowListener(listener);
    }
}   // continúa en próxima lámina
```

Ejemplo (continuación)

```
class MyWindowListener implements WindowListener {
   // Do nothing methods required by interface
   public void windowActivated( WindowEvent e) {}
   public void windowDeactivated( WindowEvent e) {}
   public void windowIconified( WindowEvent e) {}
   public void windowDeiconified( WindowEvent e) {}
   public void windowOpened( WindowEvent e) {}
   public void windowClosed( WindowEvent e) {}

   // override windowClosing method to exit program
   public void windowClosing( WindowEvent e) {
      System.exit( 0); // normal exit
   }
}
class Main {
   public static void main( String[] args) {
      CloseableFrame f = new CloseableFrame();
      f.setVisible(true);        // make the frame visible
   }  // Here the program does not end, it enters the infinite loop,
}     // because we created a graphics object, a JFrame.
```

OJO: una opción habría sido declarar esta clase como interna a Closeable. Pudo ser anónima.

Relación estática de clases (generada con Jprasp)

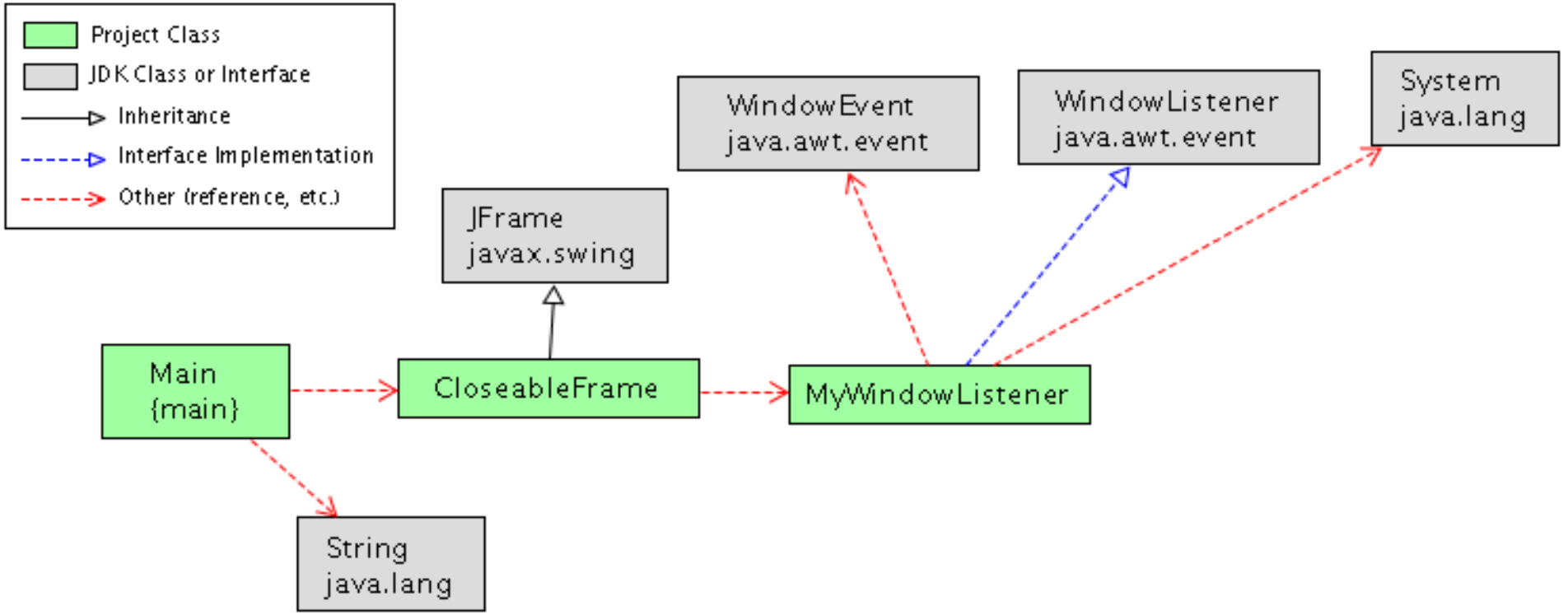
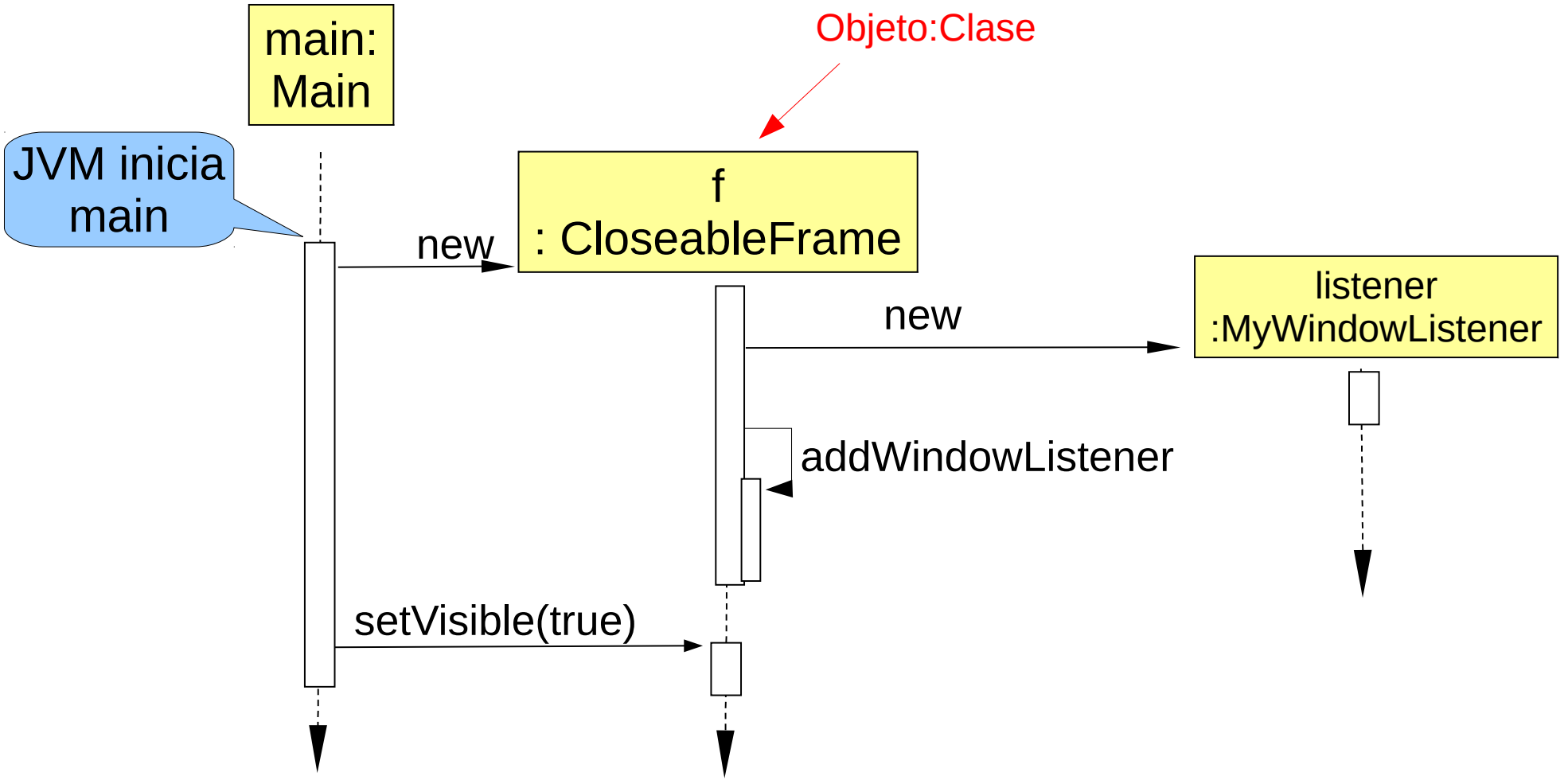


Diagrama de secuencia UML para creación de ventana

- Muestra la dinámica al crear una ventana.



Explicación

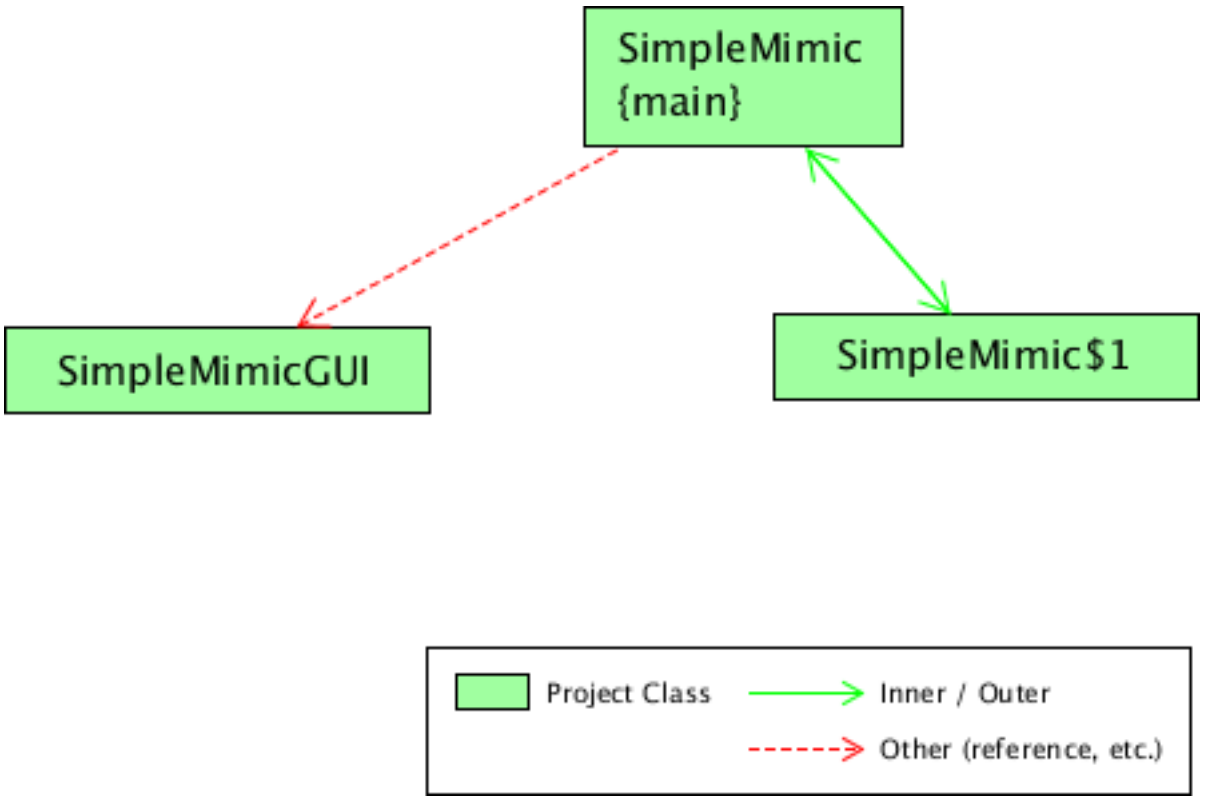
- Objetos en la clase CloseableFrame causan que una ventana aparezca en la pantalla del usuario. La aplicación puede crear tantas ventanas como lo desee creando múltiples objetos CloseableFrame.
- Una instancia de MyWindowListener es registrada con addWindowListener (es como configurar quién atenderá los eventos de la ventana). Cuando el evento ocurre, la máquina virtual Java lo detecta y verifica si hay listeners esperando por ese evento. Si los hay, automáticamente llama al método apropiado de la interfaz WindowListener según el evento ocurrido.
- En este ejemplo, la interfaz WindowListener es implementada por la clase MyWindowListener, así su instancia puede responder a los eventos de la ventana en que fue registrado el objeto.
- Hay siete métodos en la Interfaz WindowListener (Ver API). Aquí sólo nos interesamos por el evento cierre de ventana.
- La mayoría de las otras interfaces para eventos no difieren mucho. Veremos otro ejemplo.

Entrada en Campo de texto

- Supongamos que queremos leer lo ingresado en un campo de texto y luego copiarlo en un texto de la ventana (label).
- Ver SimpleMimic.java

Diagrama de Clases de SimpleMimic

Generado con Jgrasp



- Este diagrama puede ser completado con las clases del JDK usadas.

Diagrama de Clases de SimpleMimic

Generado con Jgrasp, con clases del JDK

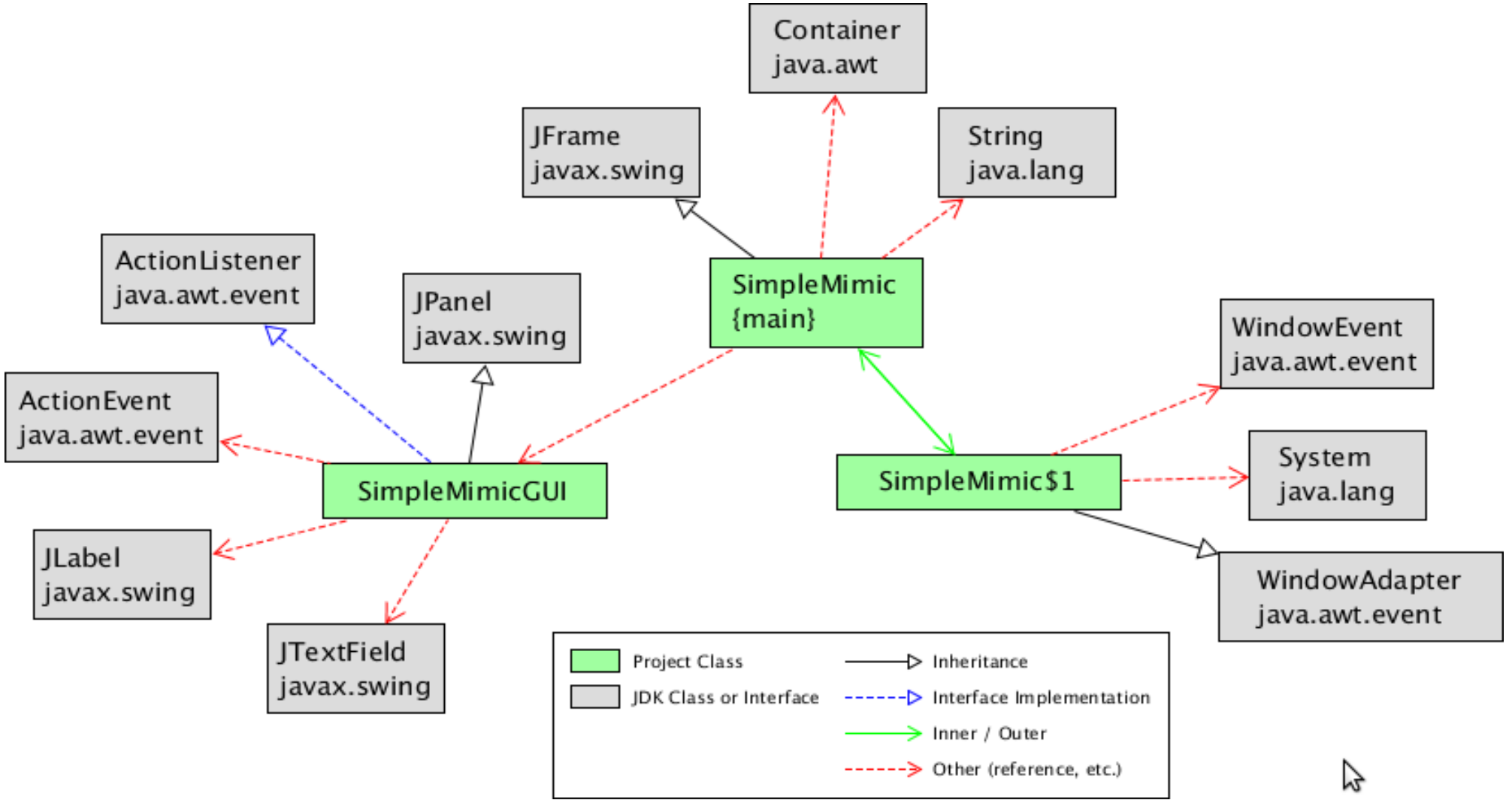
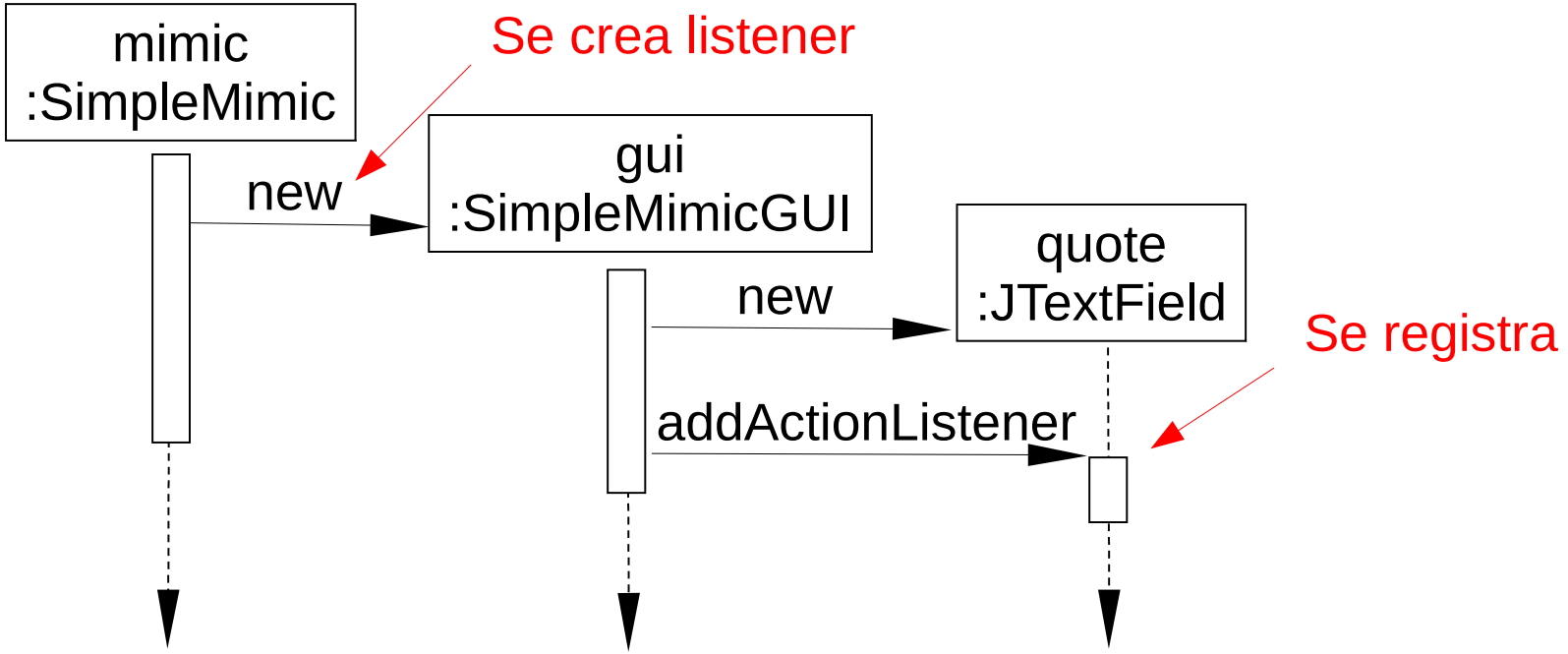


Diagrama de secuencia

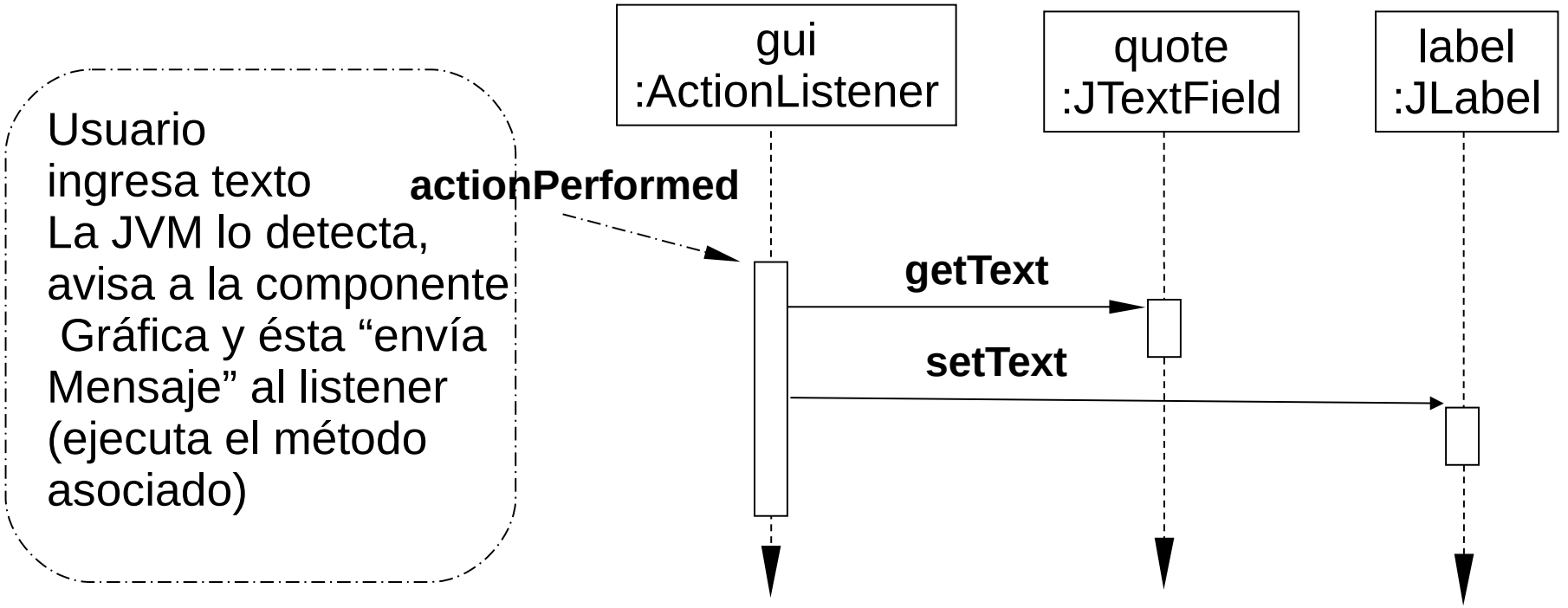
- **Situación**: Creación de listener (manejador del evento) y su registro en la componente gráfica (JTextField).



- Se espera que este tipo de diagramas puedan ser hechos antes de escribir el código (alguien experimentado).
- Estos diagramas son una buena documentación del código.

Diagrama de secuencia

- **Situación** (caso de uso): Usuario ingresa nuevo texto y presiona “Enter”. Esto gatilla el evento esperado.



- Es recomendable tener bien clara esta interacción de objetos. Este diagrama sirve además como documentación.
- JTextField y JLabel pertenecen a la API, luego solo hay que escribir el código para las otras tres clases.

Entrada en Campo de texto: versión 2

- Esta versión separa roles -Panel y listener-, para este problema simple probablemente no se justifica.
- La idea es explorar otras situaciones posibles.
- Ver Mimic.java

Diagrama de Clases de Mimic.java

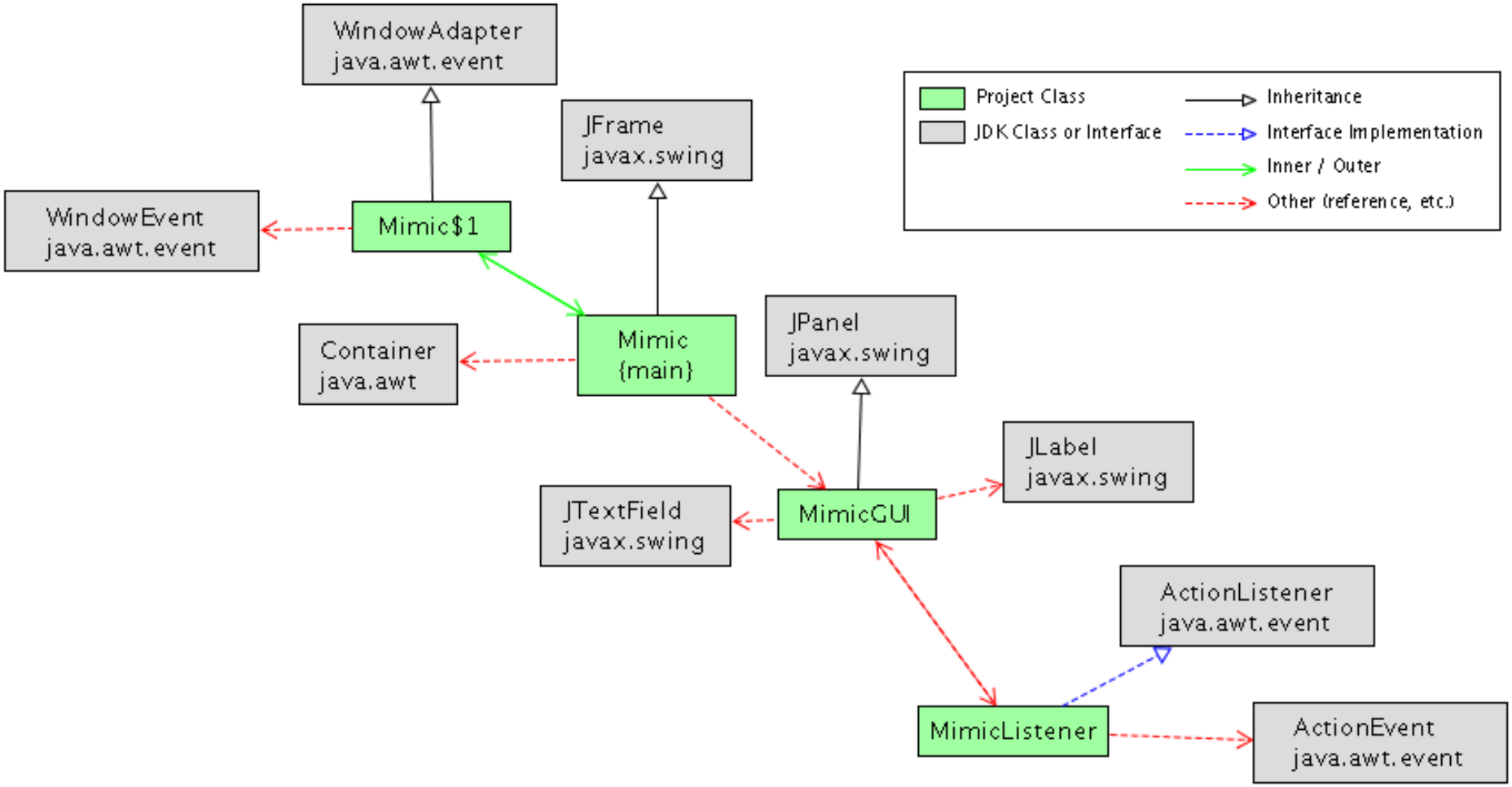
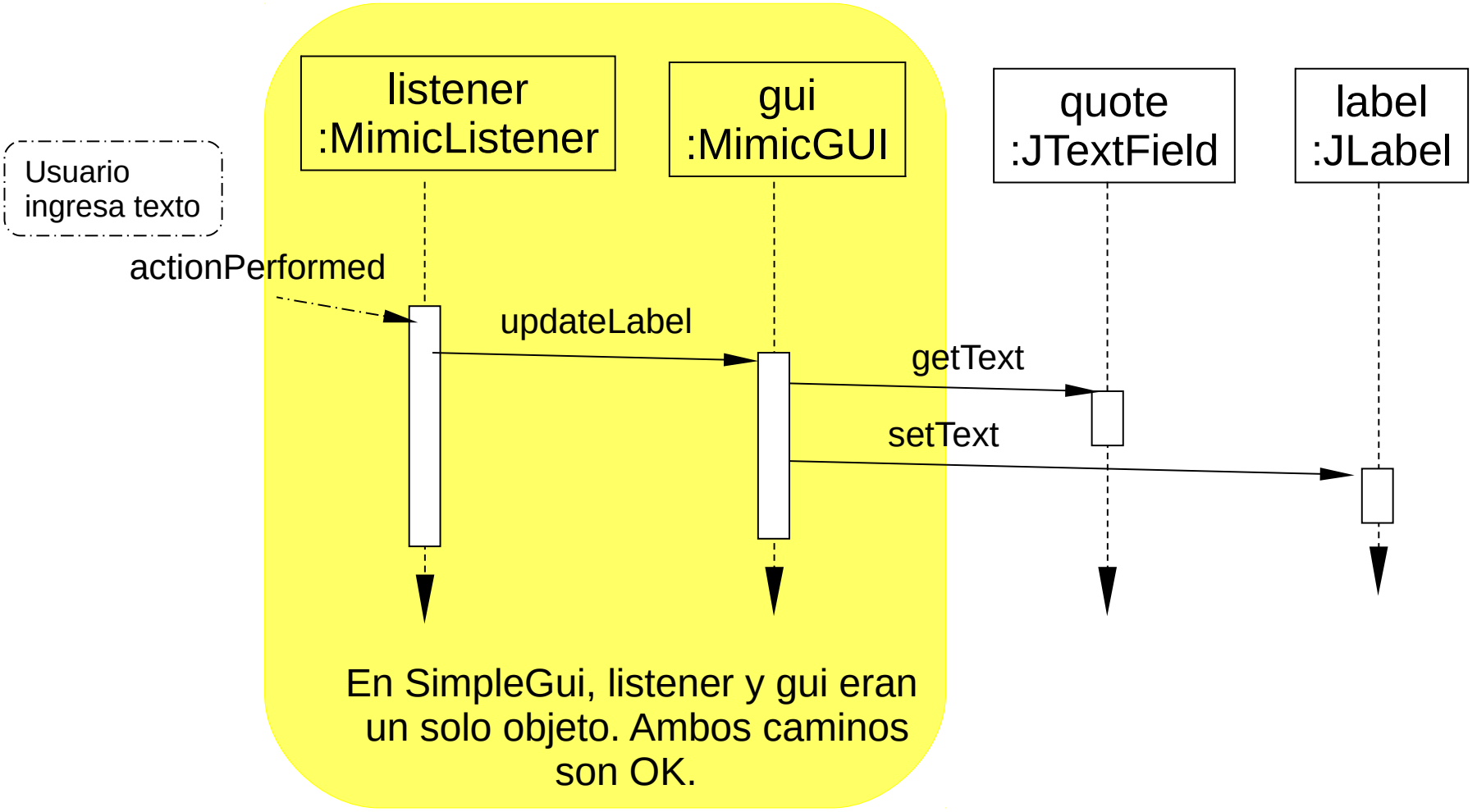


Diagrama de secuencia

- Caso de uso: Ingreso de nuevo texto. Esto gatilla el evento esperado.



Explicación del ejemplo

- El listener es registrado con el objeto quote de la clase JTextField en el constructor MimicGUI() al ejecutar quote.addActionListener(listener).
- Cuando listener es registrado, éste es agregado a una lista interna que mantiene los objetos que deben ser notificados cuando ocurre un evento. Podemos tener más de un listener registrado por evento.
- Cuando el evento ocurre (por ejemplo, presionamos return en la ventana), el método listener.actionPerformed(ActionEvent event) es llamado por el objeto JTextField, el cual se entera del evento por la JVM. Notar que los datos sobre el evento son pasados al método vía el parámetro.
- El código en el método del listener maneja el evento llamando a gui.updateLabel(), el cual copia el contenido del campo texto en el rótulo (Label) puesto en la ventana.

Algunas recomendaciones

- No es estrictamente necesario poner la descripción de las componentes de la GUI en una clase separada, pero es buena idea. Incluso puede ser conveniente poner cada una en una archivo separado para así distinguir la presentación e interacción con el usuario del procesamiento o cálculo interno.
- La clase JTextField es incluso más completa. Nos permite recibir una notificación cada vez que un caracter es ingresado.
- Los cambios requeridos para ello se muestras en MimicCharbyChar.java