Azzolini Riccardo

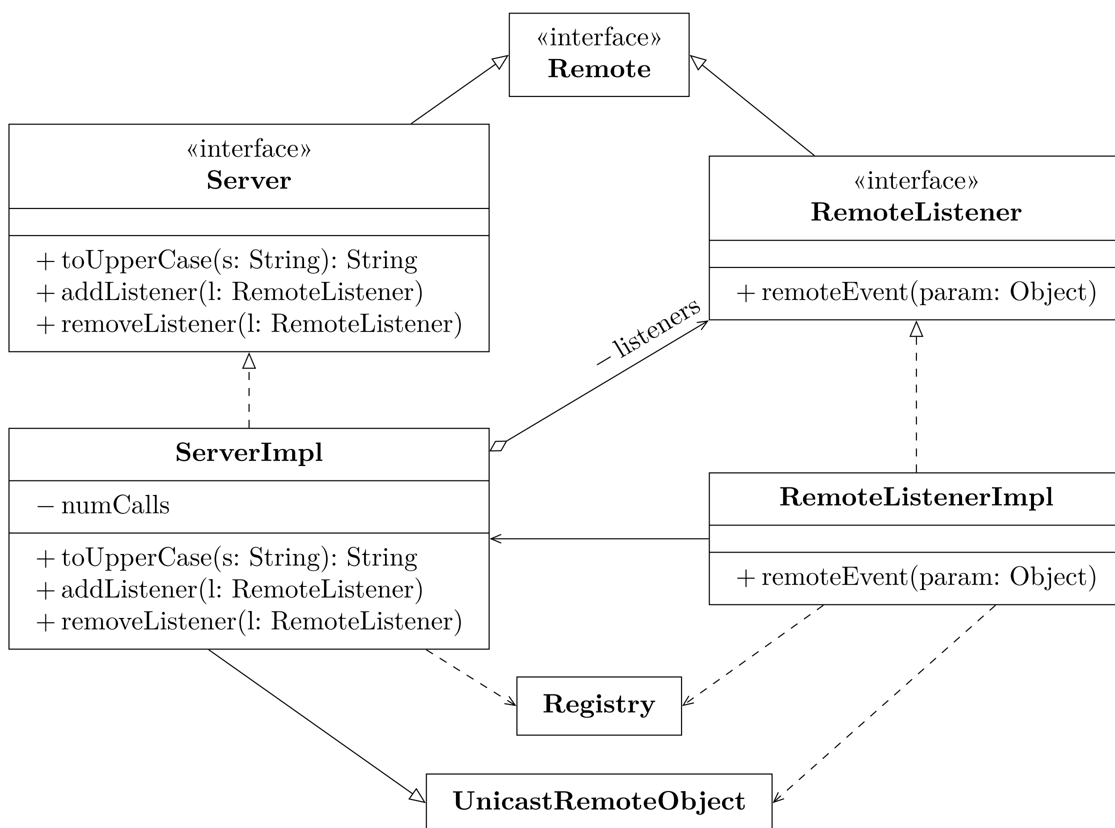
# Gestione degli eventi

## 1 Invio di notifiche ai client

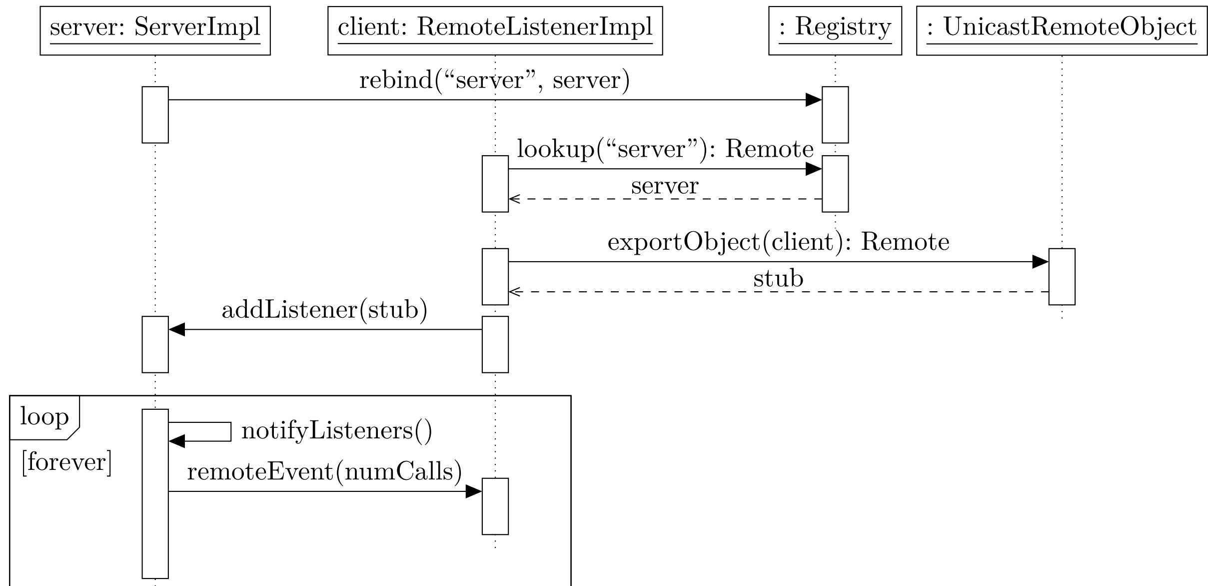
Come visto in precedenza, RMI permette l'invocazione reciproca di metodi tra due oggetti remoti "client" e "server" attraverso la tecnica del *callback*.

Un uso tipico di questa funzionalità è quello di permettere al server di inviare ai client delle notifiche (quando necessario, in base alla logica applicativa), mentre i client possono continuare normalmente a richiedere servizi al server.

Un class diagram che illustra un esempio di tale situazione è il seguente, del tutto analogo a quelli già mostrati negli esempi di callback:



Anche il sequence diagram che mostra un esempio di comportamento del sistema è uguale al solito, per quanto riguarda la parte di inizializzazione (nella quale server e client acquisiscono l'uno il riferimento remoto dell'altro), ma la "novità" è ciò che accade dopo: periodicamente, in un ciclo infinito (nel caso di questo esempio), il server invia una notifica al client, con la quale comunica (a scopo illustrativo) il numero di richieste di servizio che ha ricevuto fino a quel momento.




## 1.1 Interfaccia del server

Il codice che definisce l'interfaccia remota del server è:

```java
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Server extends Remote {
    public String toUpperCase(String s) throws RemoteException;
    public void addListener(RemoteListener l) throws RemoteException;
    public void removeListener(RemoteListener l) throws RemoteException;
}
```

- Il metodo `toUpperCase` corrispondente al servizio "vero e proprio" offerto dal server (qui è un semplice servizio che riceve una stringa e restituisce la stessa stringa con tutti i caratteri resi maiuscoli).
- `addListener` permette a un client di registrarsi presso il server, inviando il proprio riferimento remoto, al fine di ricevere notifiche dal server.

- `removeListener` chiede al server di "dimenticare" il riferimento remoto di un client, e quindi di non inviare più notifiche a quest'ultimo.

### 1.2 Interfaccia del client

L'interfaccia remota del client prevede un singolo metodo, `remoteEvent`, che il server chiama per inviare una notifica al client, tipicamente al fine di comunicare che è avvenuto un qualche evento:

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RemoteListener extends Remote {
    public void remoteEvent(Object param) throws RemoteException;
}
```

Il parametro di `remoteEvent` è di tipo `Object`, cioè può contenere qualunque oggetto: perciò, questo singolo metodo potrebbe essere sufficiente per passare qualunque informazione al client (ma, normalmente, si preferisce usare più metodi specifici).

### 1.3 Implementazione del server

Il server mantiene una lista di riferimenti remoti ai client registrati (`listeners`) e un conteggio del numero di chiamate del metodo `toUpperCase`, il cui valore viene periodicamente inviato (mediante l'invocazione remota del metodo `remoteEvent`) a tutti i client registrati:

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.*;

public class ServerImpl extends UnicastRemoteObject implements Server {
    private final List<RemoteListener> listeners = new ArrayList<>();
    private int numCalls = 0;

    public ServerImpl() throws RemoteException {}

    public String toUpperCase(String s) {
        synchronized (this) { numCalls++; }
        return s.toUpperCase();
    }
```

```java
    public synchronized void addListener(RemoteListener l) {
        listeners.add(l);
    }

    public synchronized void removeListener(RemoteListener l) {
        listeners.remove(l);
    }

    private synchronized void notifyListeners() {
        Iterator<RemoteListener> iter = listeners.iterator();
        while (iter.hasNext()) {
            try {
                RemoteListener l = iter.next();
                l.remoteEvent(numCalls);
            } catch (RemoteException e) {
                // Il client si è probabilmente disconnesso
                iter.remove();
            }
        }
    }

    public static void main(String[] args)
            throws InterruptedException, RemoteException {
        ServerImpl server = new ServerImpl();
        System.out.println("Registering...");
        Registry registry = LocateRegistry.createRegistry(1099);
        registry.rebind("server", server);
        System.out.println("Registered");
        while (true) {
            server.notifyListeners();
            Thread.sleep(500);
        }
    }
}
```

### 1.4 Implementazione del client

Il client fa qualche richiesta di servizio al server, e, intanto, stampa tutte le notifiche ricevute da quest'ultimo:

```java
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
```

```java
import java.rmi.server.UnicastRemoteObject;

public class RemoteListenerImpl implements RemoteListener {
    public void remoteEvent(Object param) {
        System.out.println("REMOTE NOTIFICATION: num calls = " + param);
    }

    private void work() throws Exception {
        Registry registry = LocateRegistry.getRegistry();
        Server server = (Server) registry.lookup("server");
        RemoteListener stub = (RemoteListener)
            UnicastRemoteObject.exportObject(this, 3939);

        server.addListener(stub);
        for (int i = 0; i < 3; i++) {
            System.out.println(
                "Remote call: " + server.toUpperCase("test " + i)
            );
            Thread.sleep(1000);
        }
        server.removeListener(stub);
        UnicastRemoteObject.unexportObject(this, false);
    }

    public static void main(String[] args) throws Exception {
        RemoteListenerImpl client = new RemoteListenerImpl();
        client.work();
    }
}
```

## 2 Quando notificare i client

Finora, si è visto solo il meccanismo di notifica dei client, con un esempio in cui il server decide arbitrariamente quando inviare notifiche: nello specifico, sceglie di farlo semplicemente a intervalli regolari di mezzo secondo. In realtà, ciò non ha molto senso: invece di aspettare una notifica, potrebbe essere direttamente il client a interrogare il server ogni mezzo secondo.

L'invio di notifiche dal server al client è veramente utile nel caso di interazioni *asincrone*, in cui il client non può sapere in anticipo quando si verificherà un determinato **evento** (a cui esso è interessato), e quindi non può interrogare il server al momento giusto. Allora, deve essere appunto il server a informare i client di tale evento.

# 3 Gestione degli eventi nelle interfacce grafiche

L'esempio tipico di gestione di eventi asincroni sono le interfacce grafiche: ogni volta che si digita un carattere, o si fa un click con il mouse, ecc., si genera un evento, e possono esserci diversi oggetti interessati a essere informati di tale evento per potervi reagire opportunamente.

*Osservazione*: Questo esempio è riferito ad applicazioni locali, non distribuite. In effetti, il meccanismo degli eventi e delle notifiche è del tutto ortogonale al fatto che gli elementi che interagiscono siano locali o distribuiti.

## 3.1 Gestione degli eventi in AWT

Il funzionamento generale della gestione degli eventi in AWT (Abstract Window Toolkit, uno degli ambienti più diffusi per creare interfacce grafiche in Java) è il seguente:

1. Si crea un listener, implementando un'interfaccia predefinita che varia a seconda del tipo di evento a cui si è interessati. Ad esempio:

   ```
   public class MyListener implements ActionListener {
       public void actionPerformed(ActionEvent e) {
           // Codice che reagisce all'evento...
       }
   }
   ```

2. Si registra il listener presso il componente dell'interfaccia grafica che genera gli eventi a cui si è interessati:

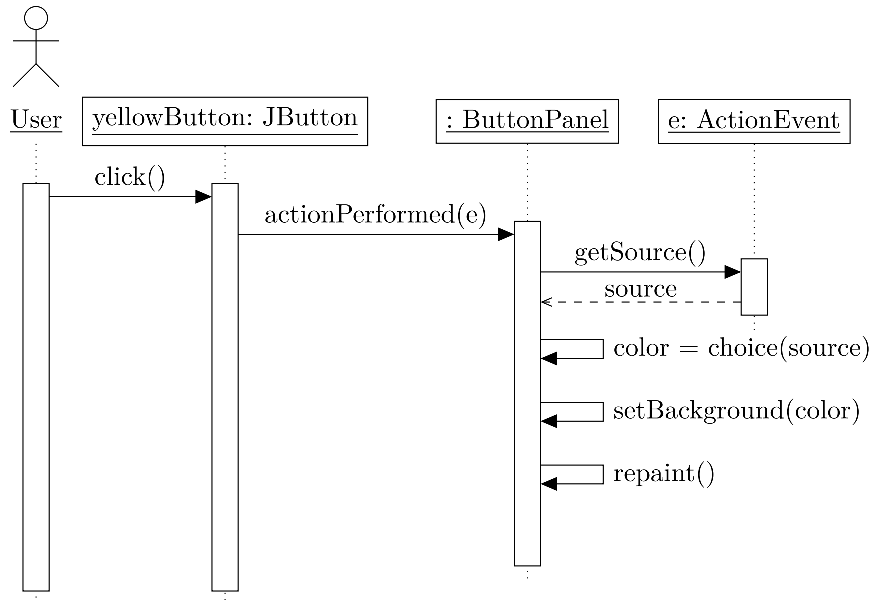
   ```
   MyListener x = new MyListener();
   someComponent.addActionListener(x);
   ```

   In pratica, il componente fa da "server", mentre il listener è il "client", che viene notificato (mediante una chiamata del suo metodo `actionPerformed`) ogni volta che si verifica l'evento.

## 3.2 Esempio: reazione al click su un bottone

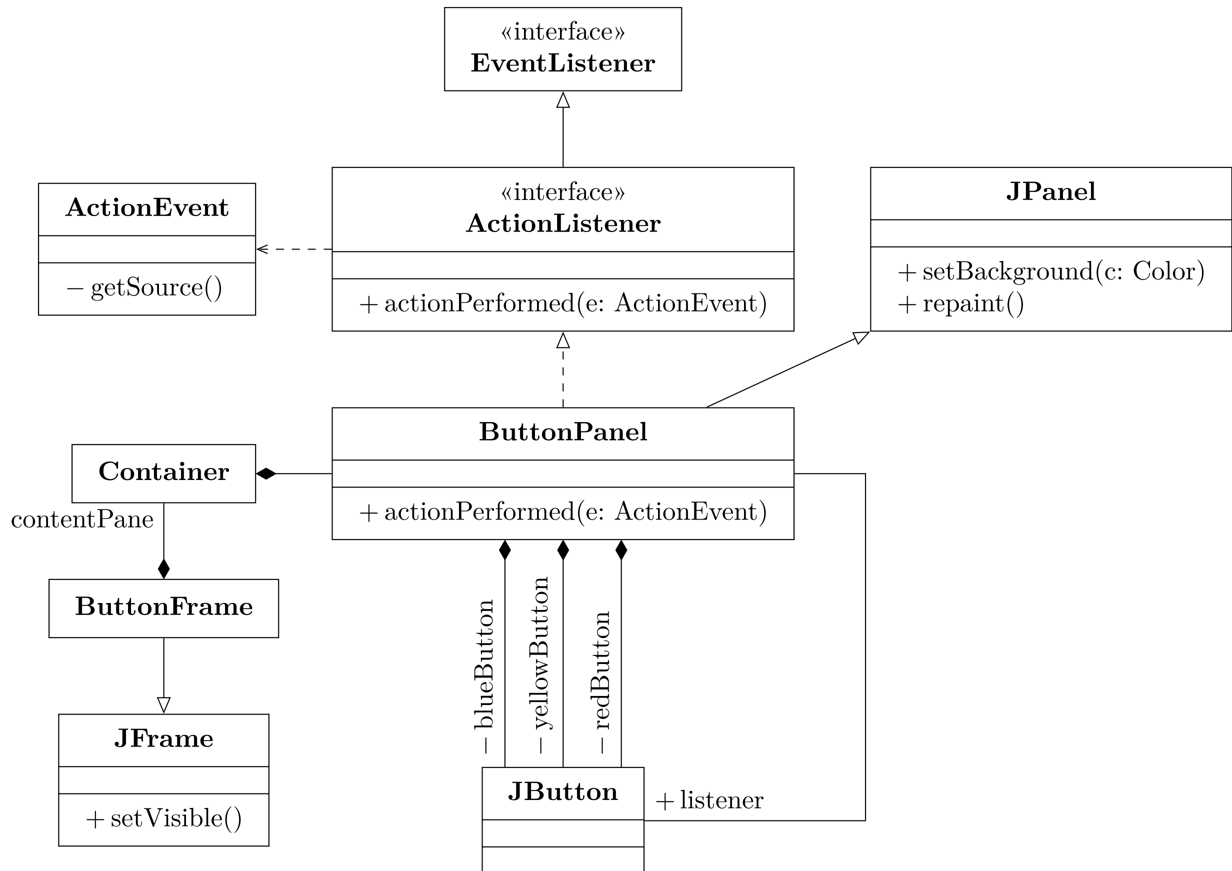
Come semplice esempio di gestione degli eventi, si vuole sviluppare un'applicazione con una finestra contenente alcuni bottoni, che permettono di cambiare il colore dello sfondo della finestra stessa.

A tale scopo, si sceglie di creare un solo listener per tutti e tre i bottoni. Un esempio di funzionamento del sistema è allora:




1. L'utente, cliccando su un bottone (qui quello corrispondente al colore giallo) genera un evento asincrono.
2. Il bottone notifica il listener (`ButtonPanel`), passando come argomento un oggetto (istanza di `ActionEvent`) che descrive l'evento.
3. Il listener usa la descrizione dell'evento per determinare sia la sorgente di quest'ultimo, ovvero quale bottone sia stato cliccato, e quindi quale sia il colore scelto. Poi, usa un apposito metodo per impostare il background della finestra, e infine chiama `repaint` per aggiornare effettivamente la visualizzazione grafica.

Il class diagram che descrive il sistema è il seguente:




La classe `ButtonPanel` funge sia da contenitore grafico per i bottoni che, come già detto, da listener:

```
import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JPanel;

public class ButtonPanel extends JPanel implements ActionListener {
    private final JButton yellowButton;
    private final JButton blueButton;
    private final JButton redButton;

    public ButtonPanel() {
        yellowButton = new JButton("Yellow");
        blueButton = new JButton("Blue");
        redButton = new JButton("Red");
```

```
        add(yellowButton);
        add(blueButton);
        add(redButton);
        yellowButton.addActionListener(this);
        blueButton.addActionListener(this);
        redButton.addActionListener(this);
    }

    public void actionPerformed(ActionEvent e) {
        Object source = e.getSource();
        Color color = getBackground();
        if (source.equals(yellowButton)) {
            color = Color.yellow;
        } else if (source.equals(blueButton)) {
            color = Color.blue;
        } else if (source.equals(redButton)) {
            color = Color.red;
        }
        this.setBackground(color);
        this.repaint();
    }
}
```

La classe `ButtonFrame` costituisce la finestra, che contiene il `ButtonPanel`:

```
import java.awt.Container;
import javax.swing.JFrame;

public class ButtonFrame extends JFrame {
    public ButtonFrame() {
        setTitle("ButtonTest");
        setSize(300, 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container contentPane = getContentPane();
        contentPane.add(new ButtonPanel());
    }
}
```

Infine, `ButtonTest` contiene il main, che semplicemente crea una finestra e la rende visibile:

```
import javax.swing.JFrame;

public class ButtonTest {
    public static void main(String[] args) {
```

```
        JFrame frame = new ButtonFrame();
        frame.setVisible(true);
    }
}
```