

Programmazione a Oggetti

Modulo B

Lezione 12

Dott. Alessandro Roncato

12/03/2013

Riassunto

- Pure Fabrication
- Simple Factory
- Abstract Factory
- Protect Variations
- Reflection

Prossime lezioni

- Lunedì 18/03: Lezione
- Martedì 19/03: Esercizi
- Lunedì 25/03: Esercizi liberi
- Martedì 26/03: 2° Compitino

Da che oggetto iniziare?

D: di che oggetto è il primo metodo chiamato?

R: gli oggetti sono “passivi” e quindi ci vuole “qualcuno” che invoca i loro metodi

Nelle applicazioni a riga di comando, c'è il metodo statico *main* che inizia la computazione

Nelle applicazioni a finestre o Web?

Controller

D: Qual è il primo oggetto (oltre lo strato delle librerie di sistema) a ricevere e coordinare un'operazione di sistema

R: ad un oggetto di invenzione che:

1) rappresenta il sistema

2) rappresenta uno scenario di un singolo caso d'uso

Opzioni

- Caso 1 (Sistema) ha senso se ci sono poche operazioni di sistema
- Caso 2 ha senso se ci sono molte operazioni di sistema
 - Ci sono due versioni:
 - 1) controller di caso d'uso: lo stesso oggetto gestisce un solo caso d'uso di tutti gli utenti
`<casoduso>Handler`
 - 2) controller di sessione: ogni attore ha il suo oggetto controller (autenticazione)
`<casoduso>Session`

Controller

- Il controller si occupa di collegare le nostre classi del modello con il resto dell'applicazione.
- Per le applicazioni con interfaccia grafica e Web esiste uno “strato” software che gestisce (per il programmatore) gli aspetti di interazione con l'utente e/o mondo esterno

Cos'è il Modello?

- In generale tutte le classi che hanno a che fare con il campo di utilizzo dell'applicazione e che NON dipendono dal tipo di interazione dell'applicazione (cioè Applicazione Web, con interfaccia grafica, a linea di comando)
- Esempio della banca: tutte le classi viste: Banca, Conto, Cliente , Operazione, etc.

Controller e basta?

- Il Controller risponde ai comandi dell'utente, ma i comandi vengono visualizzati su una interfaccia grafica.
- Chi si occupa della visualizzazione dell'interfaccia grafica?
- Nelle applicazioni a linea di comando, non è un grosso problema in quanto la visualizzazione si riduce a dei `print` e quindi può essere fatta dal `main`.

Model View Controller

D: come evitare di duplicare codice in base alle differenti modalità di presentazione (visualizzazione)

Esempio:

Gli stessi dati devo essere usati via Web, applicazione stand alone e tramite servizi esportazione/importazione (XML?)

MVC

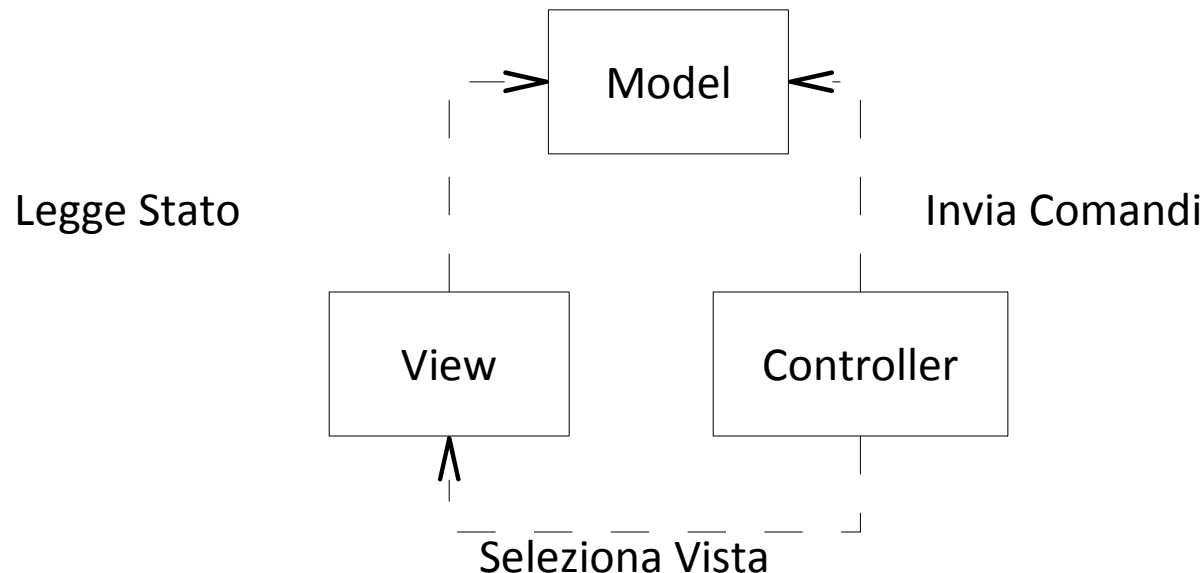
- Si separano:
 - 1) Modello
 - 2) Logica di presentazione (View)
 - 3) Logica di controllo (Controller)

Questo permette a “viste” diverse di utilizzare lo stesso modello

Il modello è indipendente da V e C!

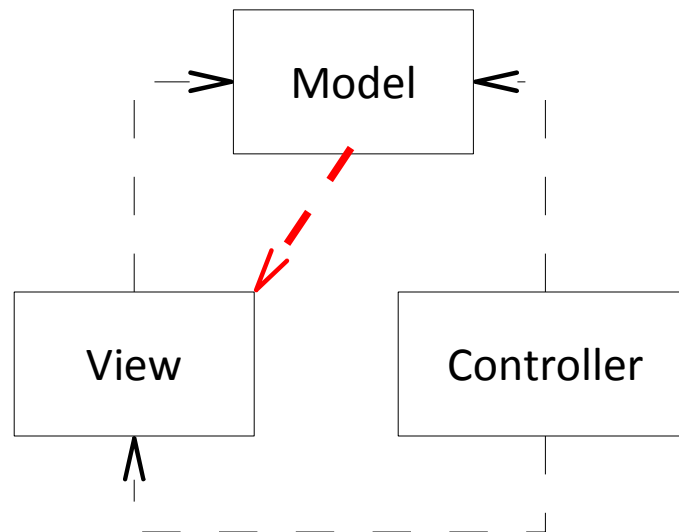
Indipendenza?

- Sia indipendenza concettuale che indipendenza con la definizione che abbiamo già dato: il Modello non usa oggetti della View e del Controller



Aggiornamento

D: come informare la View dei cambiamenti del Modello senza creare una dipendenza del modello dalla Vista?



Esempio

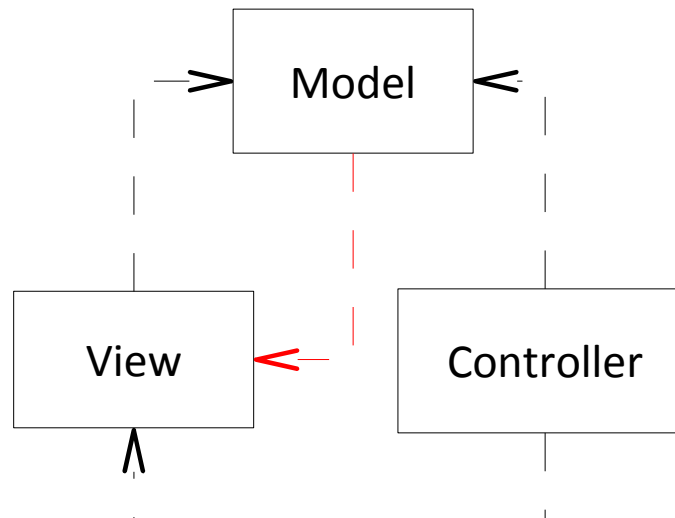
- Come fare in modo che un cambiamento dello stato del Modello visualizzato sulla “finestra” dell'applicazione?
- Se il modello richiama un metodo della View per “comunicare” il cambio di stato si crea una dipendenza del modello dalla View.

Esempio codice

```
public class Conto {  
    Importo JComponent jImporto;  
    ...  
    public void aggiornaInteressi() {  
        ...  
        jImporto.setImporto(true) ;  
        jImporto.invalidate() ;  
    }  
}
```

Esempio codice

```
public class ImportoJComponent extends JComponent {  
    double importo;  
    public void setImporto(double v){importo=v;}  
    public void paintComponent(Graphics g){  
        Graphics2D g2= (Graphics2D)g;  
        g2.drawString(""+importo,50,100);  
    }  
}
```



Svantaggi

- Abbiamo detto che non ci preoccupano le dipendenze con le classi delle librerie standard. Però in questo caso non vogliamo dipendere da `JComponent` perché in un'applicazione a linea di comando o in un'applicazione Web non è disponibile `JComponent` e quindi non sarebbe possibile riusare la classe `Conto` in un'altro tipo di applicazione.

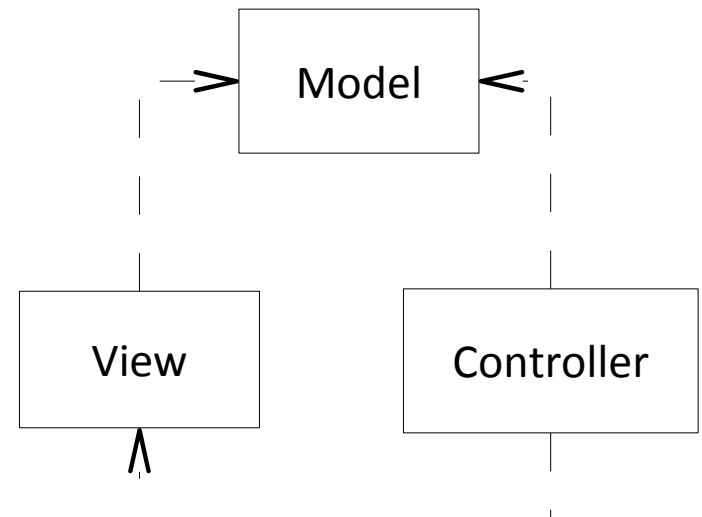
Alternativa

Fare in modo che sia il controller a gestire tutto:

- Bassa coesione
- Complessità codice
- Funziona solo se il controller **conosce** esattamente quando il modello cambia stato

Esempio alternativa

```
public class Controller {  
    ImportoJComponent jComponent;  
    Prodotto prodotto=...;  
    ...  
    public void doAction() {  
        ...  
        jComponent.setImporto(conto.getSaldo());  
        jComponent.invalidate();  
    }  
}
```



Pattern Observer

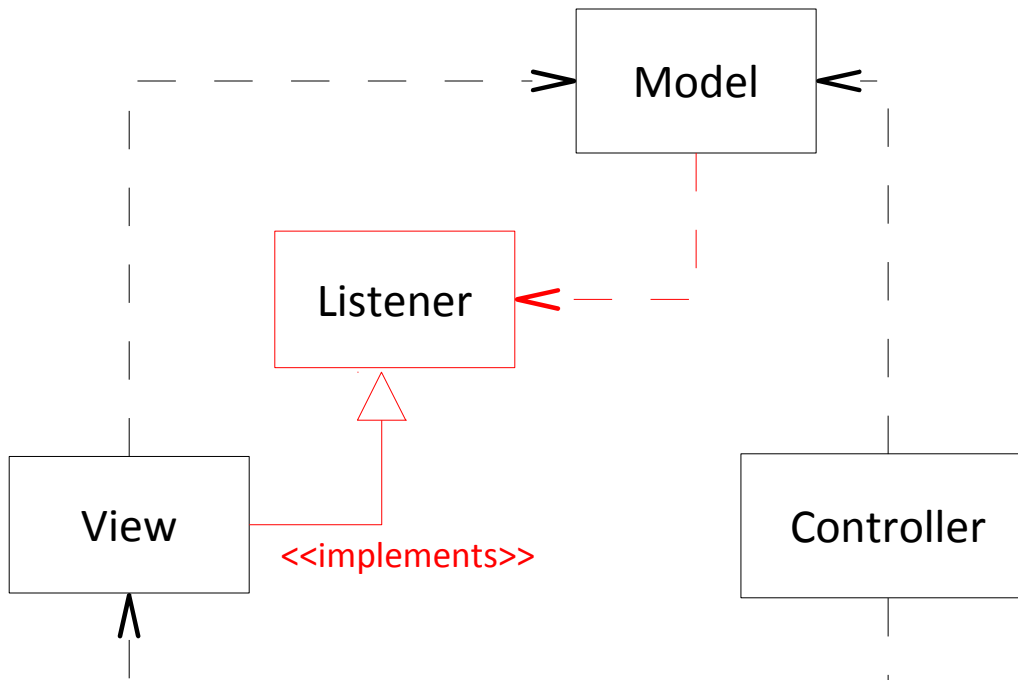
- Uno o più oggetti (detti Subscribers) sono interessati agli eventi (o cambi di stato) di un altro oggetto (detto Publisher).
- Il Publisher vuole essere quanto più indipendente dai Subscribers

Pattern Observer

Soluzione:

- 1) si definisce un'interfaccia Listener
- 2) i Subscriber implementano Listener
- 3) il Publisher registra **dinamicamente** i subscribers
- 4) il Publisher avvisa i Subscribers registrati quando si verifica l'evento

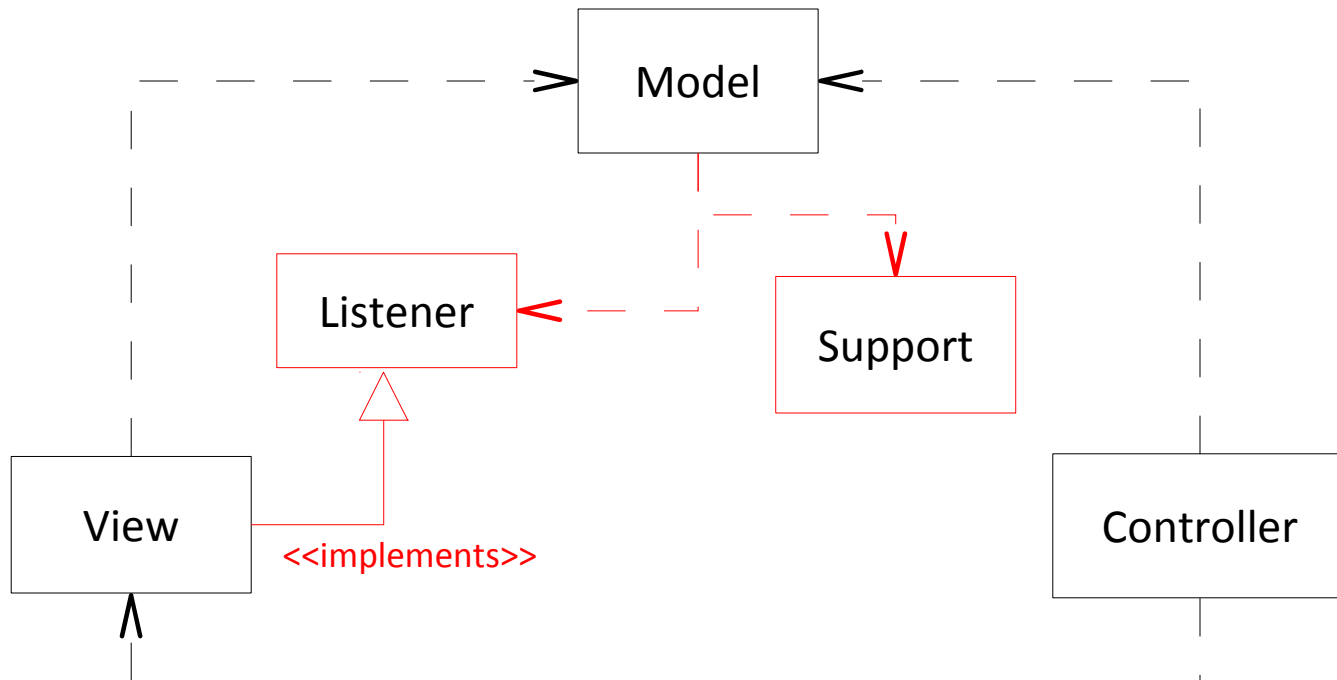
Diagramma



Observer in Java

- Già definite varie interfacce Listener (a seconda del tipo di evento che si vuole “ascoltare”)
- Già definite e implementate varie classi di supporto (ChangeSupport) per implementarle
- Stabile

Diagramma in Java



Esempio

```
public class Conto {  
    private PropertyChangeSupport listeners  
        = new PropertyChangeSupport(this);  
    ...  
    public void addPropertyChangeListener(PropertyChangeListener l){  
        listeners.addPropertyChangeListener(l);  
    }  
    public void removePropertyChangeListener(PropertyChangeListener l){  
        listeners.removePropertyChangeListener(l);  
    }  
    public void calcolaInteressi(){  
        ...  
        listeners.firePropertyChange("importo",oldValue,value);  
    }  
}
```

Esempio

```
public class ImportoComponent extends JComponent
implements PropertyChangeListener {
    double property;

    ...

    public propertyChange(PropertyChangeEvent evt) {
        property=evt.getNewValue();
        invalidate();
    }

    public void paintComponent(Graphics g) {
        Graphics2D g2= (Graphics2D)g;
        g2.drawString(""+importo,50,100);
    }
}
```

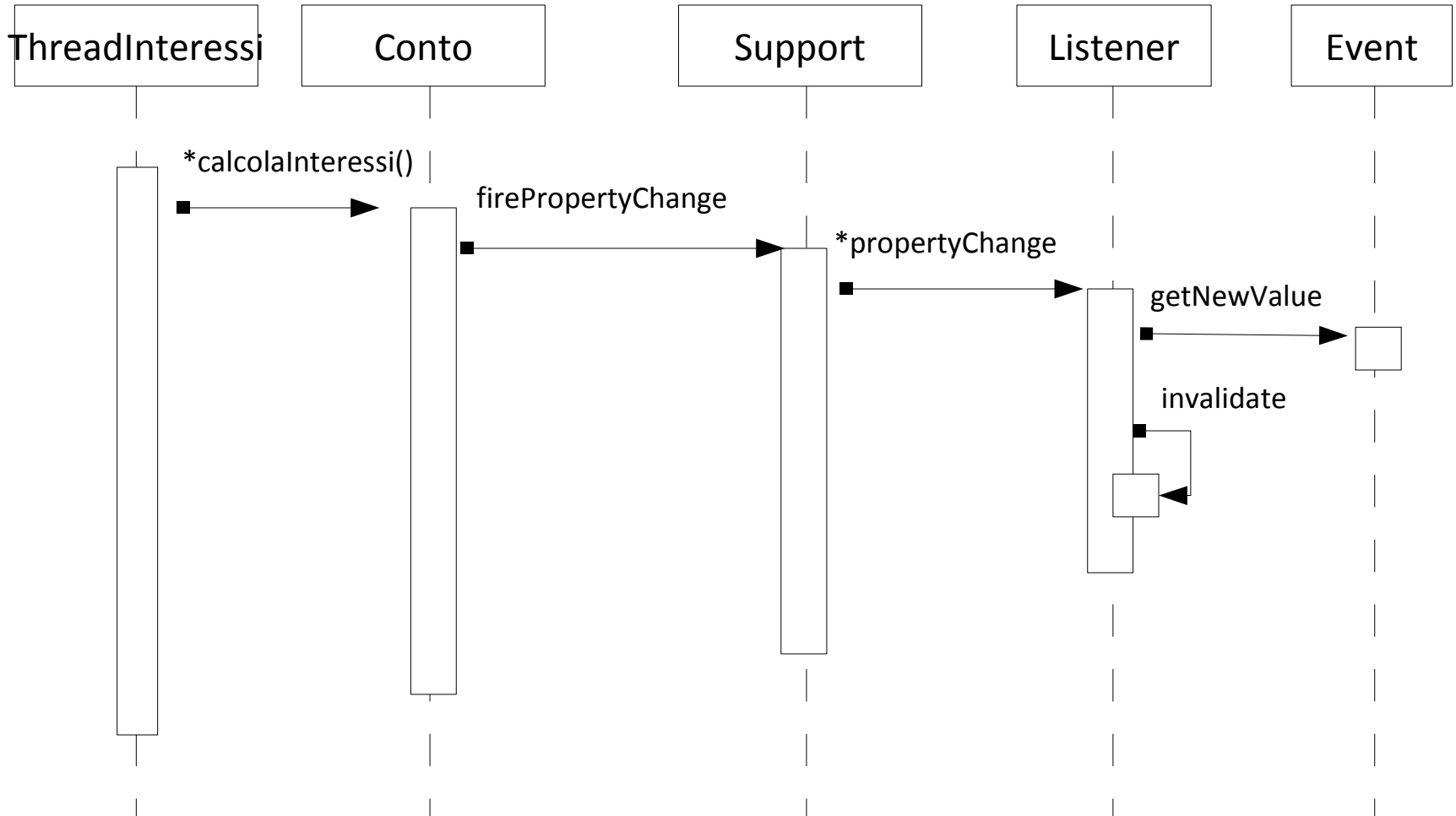
Esempio

```
public class ThreadInteressi extends Thread {  
    Set<Conto> conti;  
  
    Public ThreadInteressi() {  
        conti = Banca.getInstance().getConti();  
    }  
    ...  
    public run() {  
        ...  
        for (Conto conto: conti)  
            conto.calcolaInteressi();  
    }  
}
```

Esempio

```
public class Controllet {  
    Conto conto;  
    ImportoComponent jComponent;  
  
    public void init() {  
        ...  
        conto.addChangeListener(jComponent) ;  
    }  
    ...  
}
```

Diagramma parziale



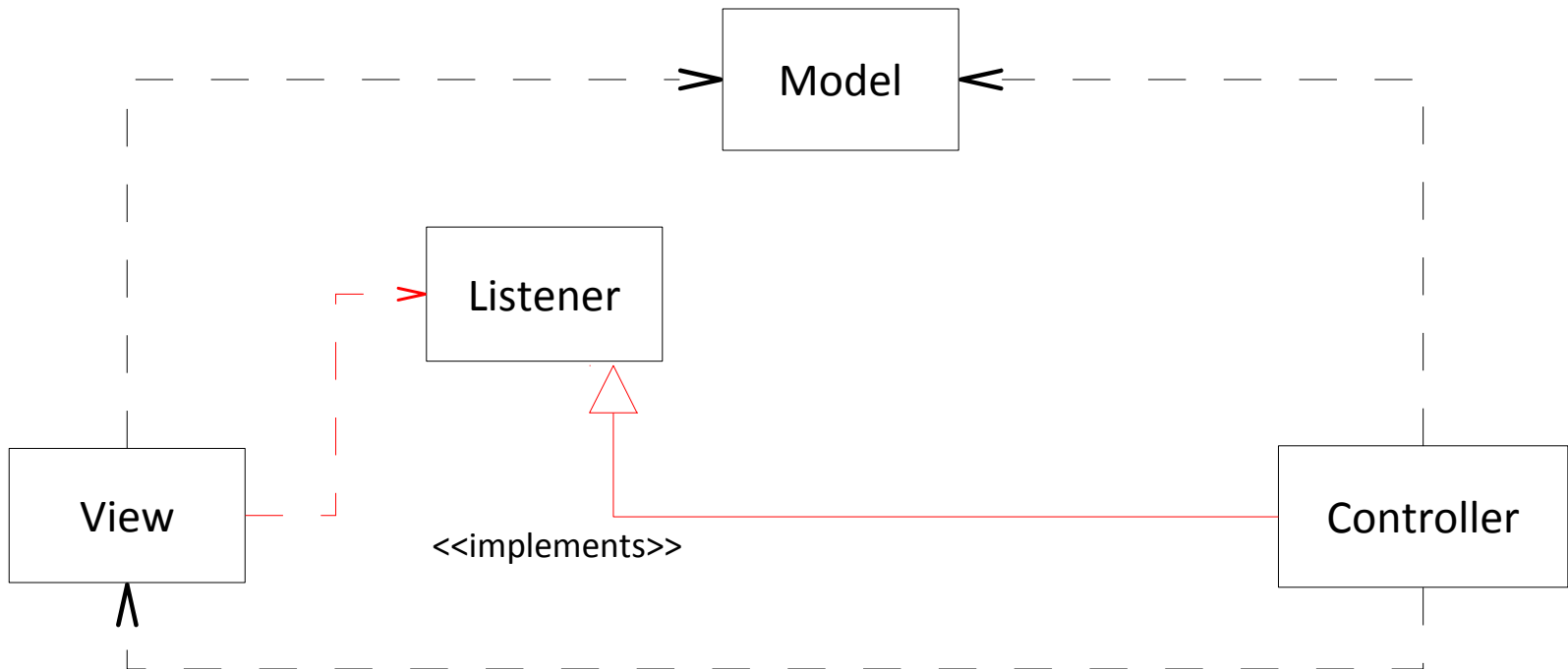
Listener e Grafica

- I listener vengono usati normalmente per notificare le interazioni dell'utente con l'applicazione (tramite l'interfaccia grafica)
- Supponiamo che un utente preme un pulsante che sta nell'interfaccia grafica, come viene invocato il metodo opportuno?

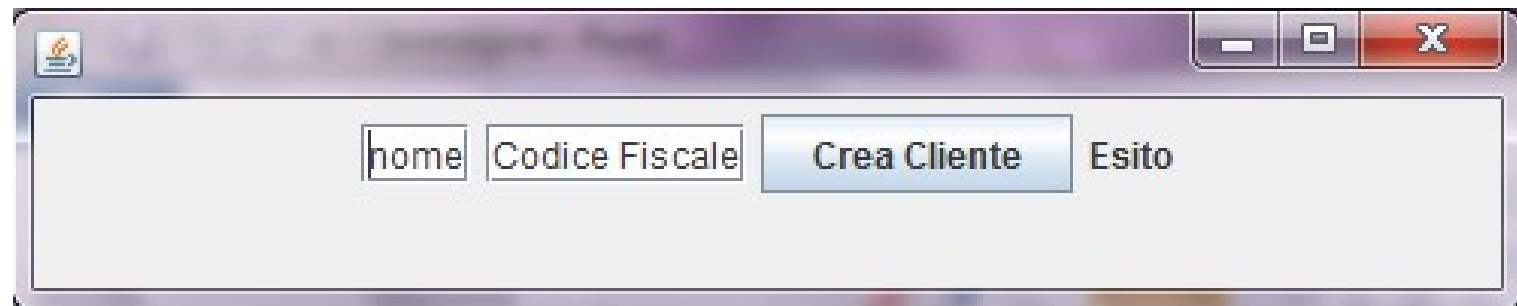
Listener e Grafica (2)

- Prima si registrano come Listener gli oggetti interessati
- Alla pressione del pulsante, tutti i Listener vengono “informati” della pressione stessa dal pulsante stesso
- “informati” significa che viene invocato il metodo opportuno del Listener

Diagramma



Esempio



home Codice Fiscale Crea Cliente Esito

Esempio

```
public class ClienteFrame extends JFrame {
    JButton crea= new JButton("Crea Cliente");
    JTextField nome = new JTextField("nome");
    JTextField codice = new JTextField("Codice Fiscale");
    JLabel esito = new JLabel("Esito");
    public ClienteFrame(ActionListener controller){
        ...
        crea.addActionListener(controller);
    public String getCodice(){
        return codice.getText();
    }
    public String getNome(){
        return nome.getText();
    }
    public void setEsito(boolean e){
        if (!e) esito.setText("Errore nella creazione!");
    }
}
```

Esempio

```
public class CreaClienteHandler implements ActionListener {
    ClienteFrame frame;
    public CreaClienteHandler() {
        frame = new ClienteFrame(this);
    }
    public void actionPerformed(ActionEvent evt) {
        String codice = frame.getCodice();
        Cliente cliente = ...findCliente(codice);
        String nome = frame.getNome();
        ...
        if (cliente==null)
            frame.setEsito (...creaCliente(codice, nome, ...));
        else
            frame.setEsito(false);
    }
}
```

Diagramma parziale

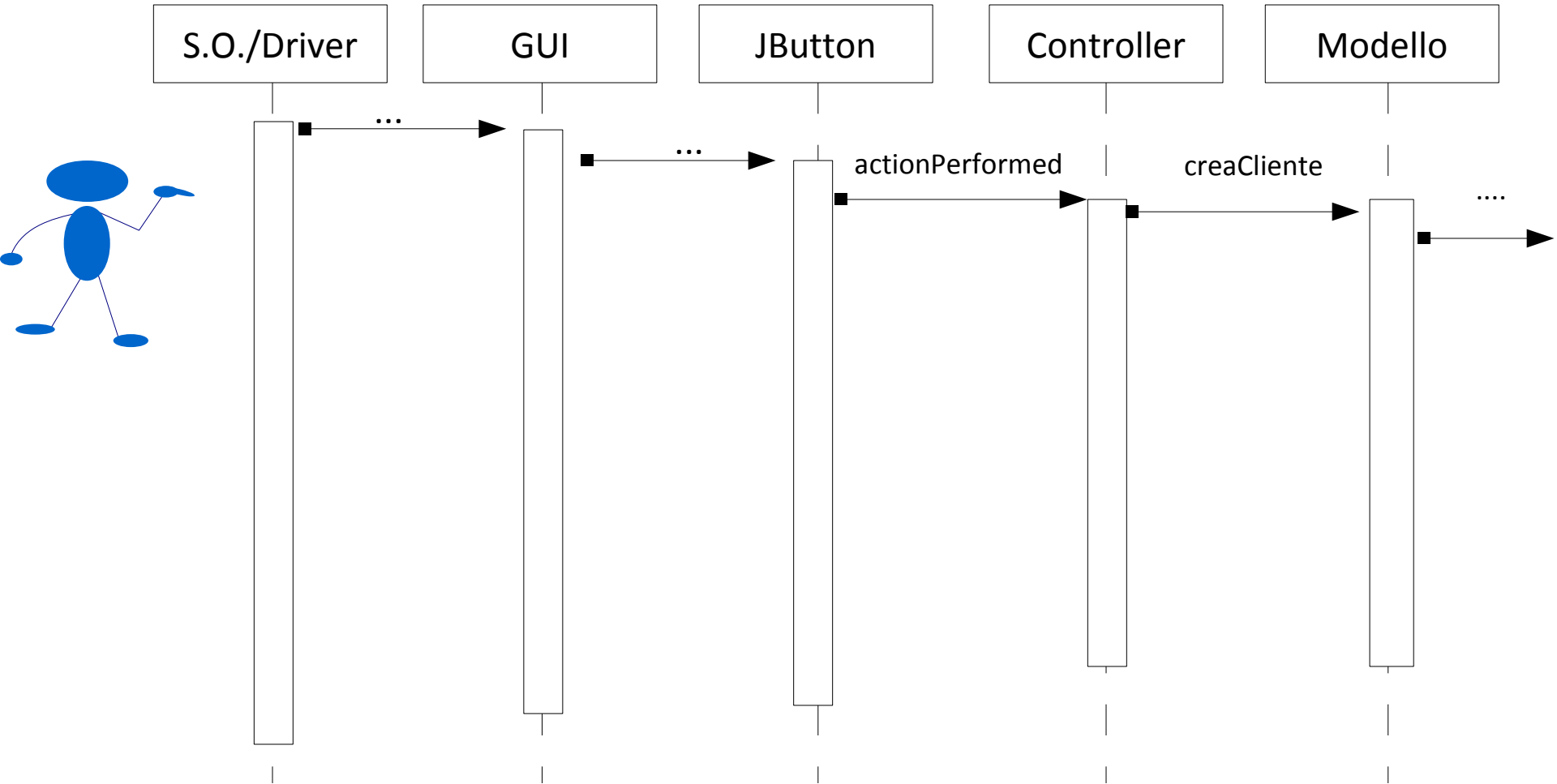


Diagramma parziale

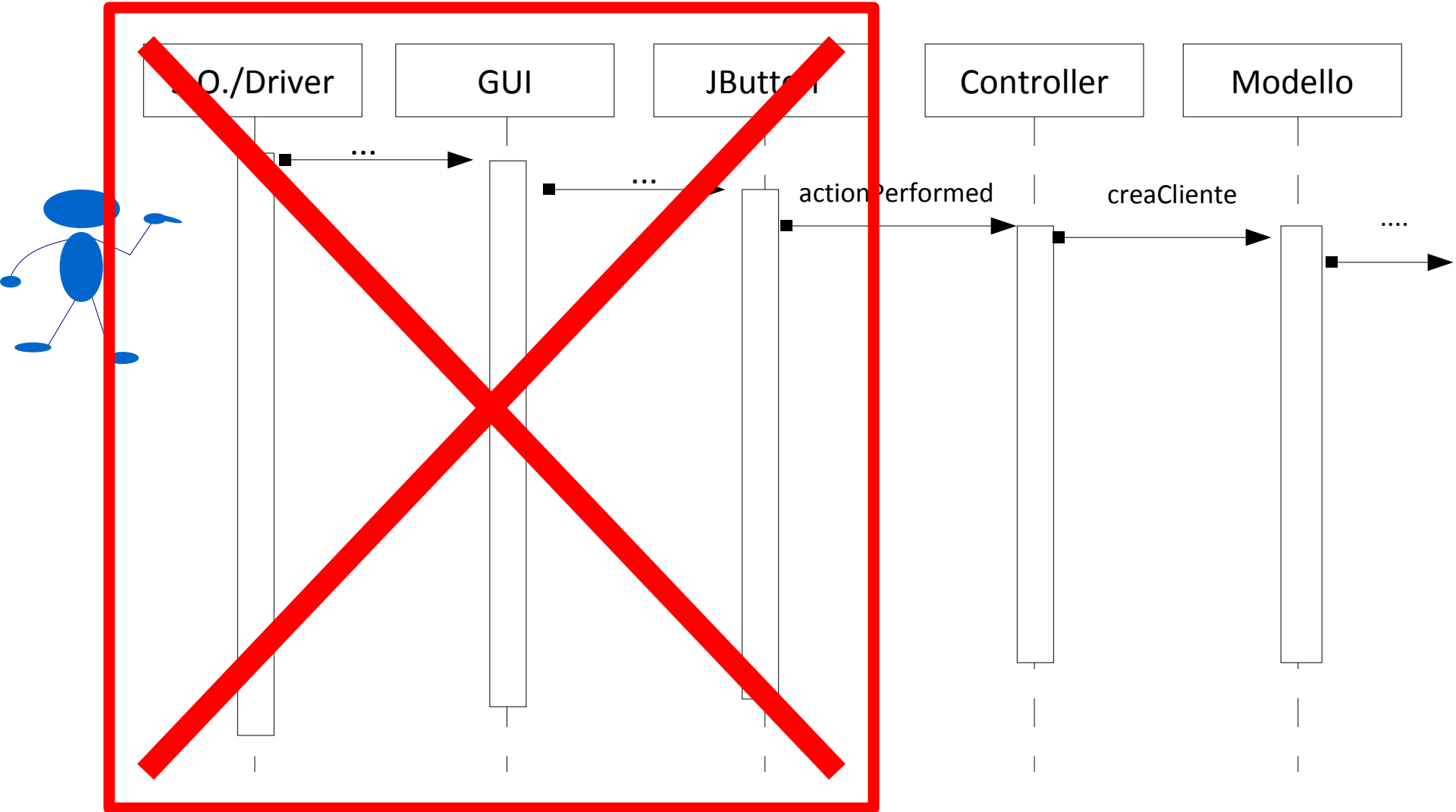
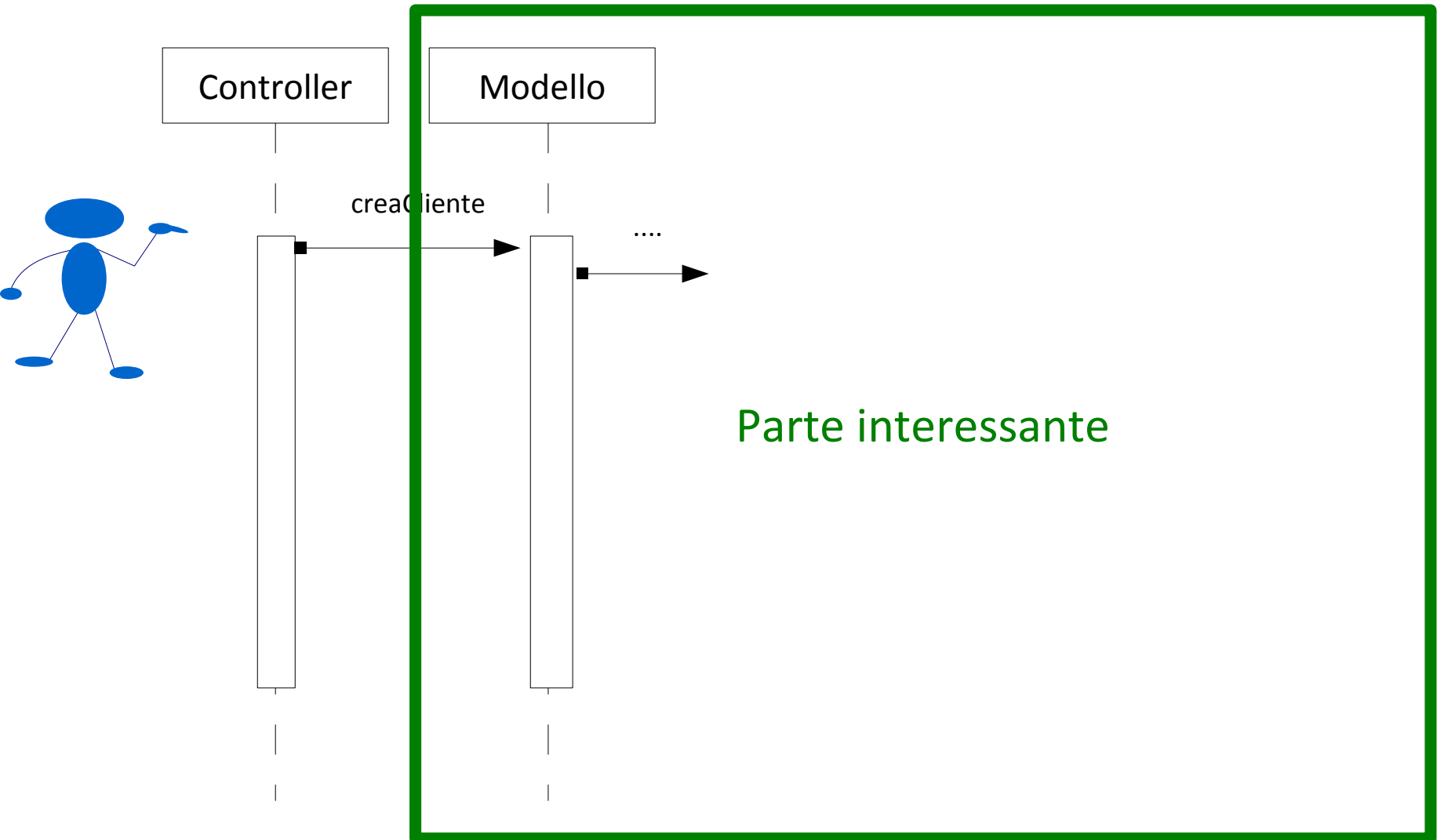


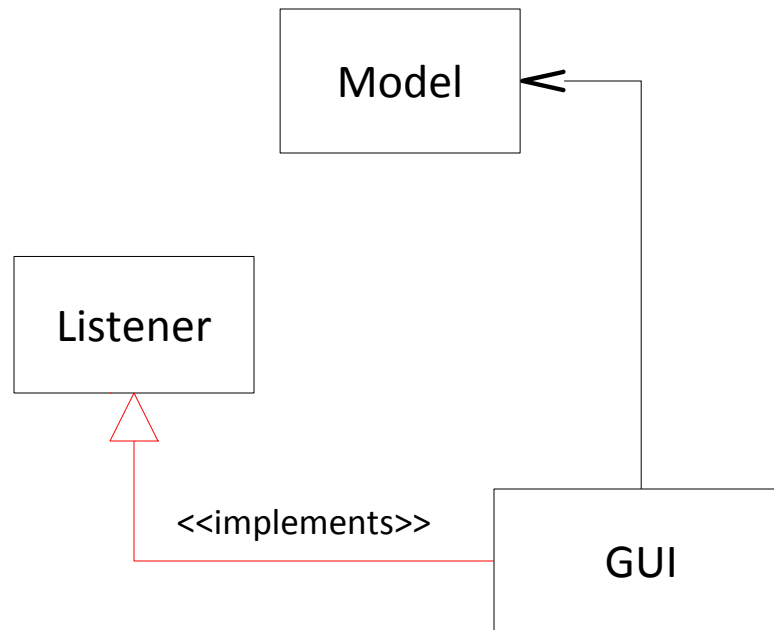
Diagramma parziale



View e Controller

- Nella API Swing generalmente le funzioni di View e Controller sono responsabilità della stessa classe (chiamata GUI) che estende un'oggetto della View con Funzioni di Controller
- Alta coesione in quanto la parte la View memorizza i dati che vengono elaborati dal Controller

Diagramma



Esempio

```
public class ClienteGUI extends JFrame implements ActionListener {  
    JButton crea= new JButton("Crea Cliente");  
    JTextField codice = new JTextField("Codice Fiscale");  
    JTextField nome = new JTextField("nome");  
    JLabel esito = new JLabel("Esito");  
    public ClientGUI(ActionListener controller) {  
        crea.addActionListener(this);  
        public String getCodice(){  
—— return codice.getText();}  
—— public String getNome(){  
—— return nome.getText();}  
—— public void setEsito(boolean e){  
—— if (!e) esito.setText("Errore nella creazione!");  
—— }  
    }  
    //continua
```

Esempio GUI

//continua

```
public void actionPerformed(ActionEvent evt) {
    String codice = codice.getText();
    Cliente cliente = ...findCliente(codice);
    String nome = nome.getText();
    ...
    if (cliente==null){
        if( ...creaCliente(codice, nome, ...))
            esito.setText ("cliente "+codice+" creato");
        else
            esito.setText("errore durante la creazione");
    }
    else
        esito.setText("cliente già esistente");
}
```

Listener Multipli

- Come facciamo a gestire più pulsanti con la stessa GUI?
- Creiamo più pulsanti
- Mettiamo in ascolto la stessa GUI per tutti i pulsanti
- Controlliamo nell'evento quale pulsante lo ha generato

Esempio ==

```
public class ClienteGUI extends JFrame implements ActionListener {  
    JButton crea= new JButton("Crea");  
    JButton elimina = new JButton("Elimina");  
    JLabel esito = new JLabel("");  
    public ClientGUI() {  
        crea.addActionListener(this);  
        elimina.addActionListener(this);  
        ...}  
    public void actionPerformed(ActionEvent evt) {  
        if (evt.getSource() == crea) {...}  
        else {...}  
    }  
}
```

Domande