

Esercitazione sui Design Pattern

Pattern Creazionali

Singleton

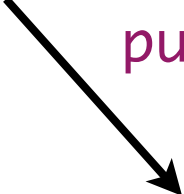
- Permette la creazione di una sola istanza della classe all'interno dell'applicazione
- Fornisce un metodo con cui ottenere l'istanza
- Il costruttore della classe non deve essere accessibile

Singleton

```
public final class Logger {  
    private final static Logger logger = new Logger();  
  
    private Logger() { ... }  
  
    public static Logger getInstance() {  
        return logger;  
    }  
  
    public void log(LoggableEvent e){}  
}
```

Singleton

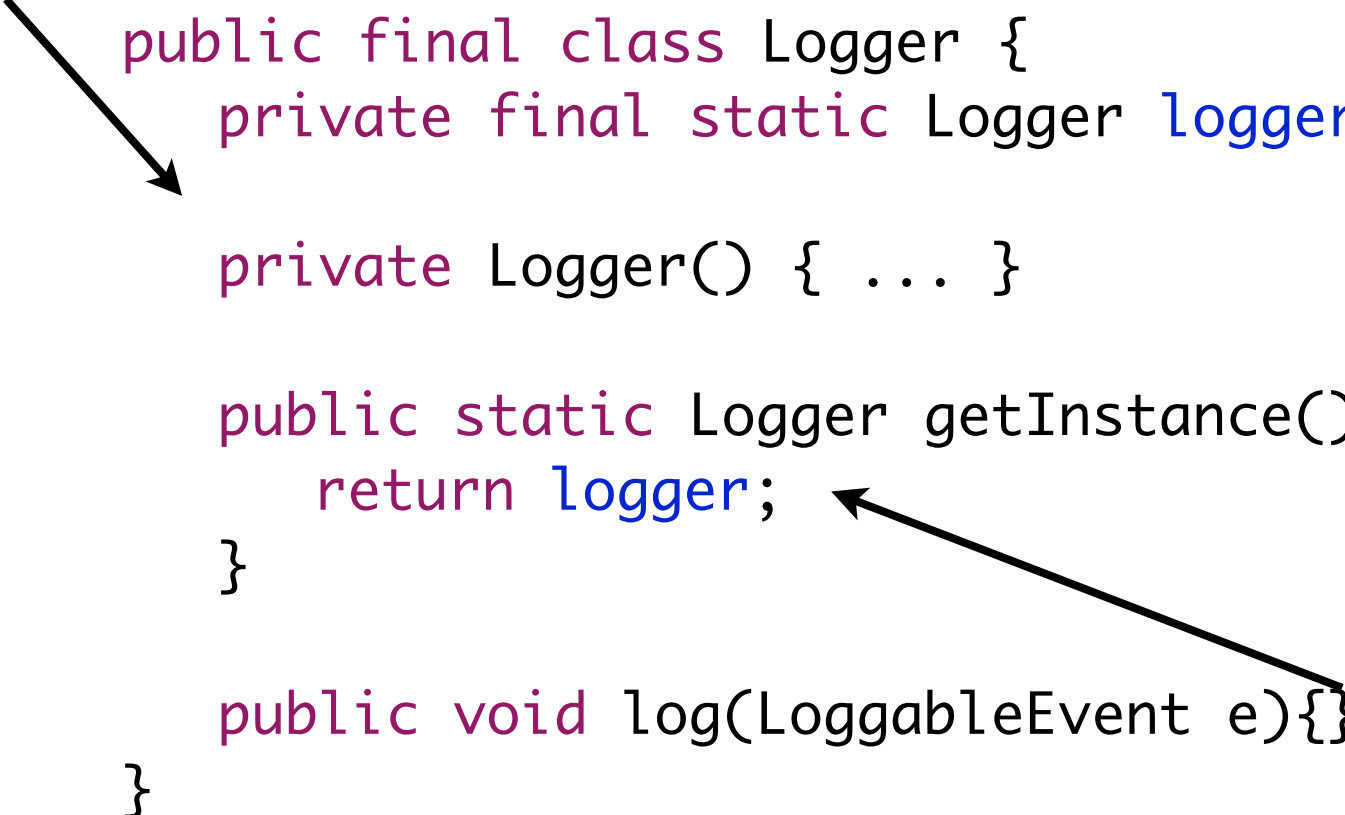
Istanza e
costruttore
privati



```
public final class Logger {  
    private final static Logger logger = new Logger();  
  
    private Logger() { ... }  
  
    public static Logger getInstance() {  
        return logger;  
    }  
  
    public void log(LoggableEvent e){}  
}
```

Singleton

Istanza e
costruttore
privati



```
public final class Logger {  
    private final static Logger logger = new Logger();  
  
    private Logger() { ... }  
  
    public static Logger getInstance() {  
        return logger;  
    }  
  
    public void log(LoggableEvent e){}  
}
```

The diagram illustrates the Singleton pattern implementation. An arrow points from the text 'Istanza e costruttore privati' to the private constructor 'private Logger() { ... }'. Another arrow points from the text 'Metodo statico per accedere all'istanza' to the 'return logger;' line within the 'getInstance()' method.

Metodo statico per
accedere all'istanza

Singleton

Istanza e
costruttore
privati

```
public final class Logger {  
    private final static Logger logger = new Logger();  
  
    private Logger() { ... }  
  
    public static Logger getInstance() {  
        return logger;  
    }  
  
    public void log(LoggableEvent e){}  
}
```

Metodo statico per
accedere all'istanza

Metodi dell'istanza

Singleton

```
public class Logger {  
    private static Logger logger = null;  
  
    private Logger() { ... }  
  
    public static Logger getInstance() {  
        if (logger == null)  
            logger = new Logger();  
        return logger;  
    }  
  
    public void log(LoggableEvent e){}  
}
```


Singleton

```
public class Logger {  
    private static Logger logger = null;  
  
    private Logger() { ... }  
  
    public static Logger getInstance() {  
        if (logger == null)  
            logger = new Logger();  
        return logger;  
    }  
  
    public void log(LoggableEvent e){}  
}
```

Meglio se la
costruzione del
singleton e'
costosa



Note

- Non deve essere usato per avere un riferimento *globale* a un oggetto:
 - sarebbe come una variabile globale!
 - Va usato come se fosse un costruttore
- Usatelo solo quando serve davvero!

Client

```
public class LoggableClass {  
  
    LoggableClass() {}  
  
    public void myMethod(...) {  
        Logger.getInstance().log(..);  
    }  
  
    public void anotherMethod(...) {  
        Logger.getInstance().log(..);  
    }  
}
```

```
public class LoggableClass {  
    private final Logger logger;  
  
    LoggableClass(Logger l) {  
        this.logger = l;  
    }  
  
    public void myMethod(...) {  
        this.logger.log(..);  
    }  
  
    public void anotherMethod(...) {  
        this.logger.log(..);  
    }  
}
```

Factory

- Disaccoppia la creazione degli oggetti dall'invocazione del costruttore
- Permette di avere una **logica** che seleziona il tipo dinamico della nuova istanza
- Utili quando ci interessa avere una classe che implementi una certa interfaccia ma non ci interessa quale sia nello specifico

Esempio

- Abbiamo diversi tipi di bottoni, ciascuno specifico per un particolare sistema operativo
- A noi interessa solamente che venga costruito un bottone
- Vogliamo gestire nel modo più pulito possibile la creazione del bottone

Soluzione ‘brutta’

- Creazione della nuova istanza (chiamata costruttore) gestita dal chiamante
- Costruttore deve sapere tipo a runtime
- La logica di creazione dell’oggetto viene mischiata con quella del suo utilizzo
- Possibili duplicazioni di codice. Va ripetuto ogni volta che ci serve un bottone

Factory

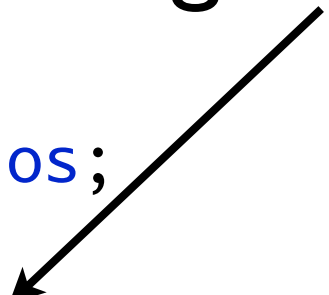
- La factory si occupa di istanziare il bottone del tipo corretto
- Possiamo separare il codice di selezione del tipo e creazione dell'istanza dal codice che deve 'usarla'

Factory

```
public class ButtonFactory {  
    private final SupportedOperatingSystems os;  
  
    public ButtonFactory(SupportedOperatingSystems os){  
        this.os = os;  
    }  
  
    public Button createButton() {  
        switch (os) {  
            case Windows:  
                return new WindowsButton();  
            case Linux:  
                return new LinuxButton();  
        }  
        return null;  
    }  
}
```


Factory

Configurazione



```
public class ButtonFactory {  
    private final SupportedOperatingSystems os;  
  
    public ButtonFactory(SupportedOperatingSystems os){  
        this.os = os;  
    }  
  
    public Button createButton() {  
        switch (os) {  
            case Windows:  
                return new WindowsButton();  
            case Linux:  
                return new LinuxButton();  
        }  
        return null;  
    }  
}
```

Factory

Configurazione

```
public class ButtonFactory {  
    private final SupportedOperatingSystems os;  
  
    public ButtonFactory(SupportedOperatingSystems os){  
        this.os = os;  
    }
```

```
    public Button createButton() {  
        switch (os) {  
            case Windows:  
                return new WindowsButton();  
            case Linux:  
                return new LinuxButton();  
        }  
        return null;  
    }  
}
```

Logica di
creazione
delle istanze

Client

```
public class Client {  
    public void useButton() {  
        ButtonFactory factory = new ButtonFactory(  
                                SupportedOperatingSystems.Windows);  
        Button button = factory.createButton();  
    }  
}
```

Client

```
public class Client {  
    public void useButton() {  
        ButtonFactory factory = new ButtonFactory(  
                                SupportedOperatingSystems.Windows);  
        Button button = factory.createButton();  
    }  
}
```

L'utilizzatore non sa (non deve/vuole sapere) qual è il tipo dinamico dell'oggetto creato



Varianti

- Esistono tantissime variazioni di factory
 - Se c'è bisogno di parametri specifici per costruire gli oggetti
 - Se questi parametri cambiano a seconda del tipo di oggetto da costruire, si può ancora rendere il codice indipendente dal tipo a runtime

Esempio

- Handler Factory (Calcolatrice)

Note

- A volte anziché creare una classe è sufficiente un metodo statico, un *factory method*

Pattern Strutturali

Decorator

- Permette di aggiungere funzionalità ad un oggetto
- Il nuovo comportamento può essere aggiunto a run time
- Non richiede la creazione di nuove sottoclassi

Esempio

- Scrivere una applicazione per rappresentare diversi tipi di pizza con diversi ingredienti
- Potremmo fare diverse sottoclassi, ma ne dovremmo fare troppe
- La soluzione delle sottoclassi non permetterebbe di comporre nuovi ingredienti a run time

Struttura base

Interfaccia
“vista” dai client

```
public interface Pizza {  
    Set<Ingredient> getIngredients();  
    float getCost();  
}
```

```
public class TomatoPizza implements Pizza {  
    @Override  
    public Set<Ingredient> getIngredients() {  
        return Collections.singleton(TOMATO);  
    }  
  
    @Override  
    public float getCost() {  
        return 5;  
    }  
}
```

Classe di base, senza
nessuna decorazione

Decorazioni

```
public abstract class PizzaDecorator implements Pizza {  
    private final Pizza decoratedPizza;  
  
    public PizzaDecorator(Pizza decoratedPizza) {  
        this.decoratedPizza = decoratedPizza;  
    }  
  
    @Override  
    public float getCost(){  
        return decoratedPizza.getCost();  
    }  
  
    @Override  
    public Set<Ingredient> getIngredients() {  
        return decoratedPizza.getIngredients();  
    }  
}
```

Contiene un oggetto (anche con altre decorazioni) e permette di aggiungergli decorazioni

Una decorazione

```
public class MozzarellaPizza extends PizzaDecorator {  
  
    public MozzarellaPizza(Pizza decoratedPizza) {  
        super(decoratedPizza);  
    }  
  
    @Override  
    public float getCost() {  
        return super.getCost() + .5f;  
    }  
  
    @Override  
    public String getIngredients() {  
        Set<Ingredient> decoratedIngredients =  
            new HashSet<Ingredient>(super.getIngredients());  
        decoratedIngredients.add(MOZZARELLA);  
        return decoratedIngredients;  
    }  
}
```

Come le usa il client

```
public class Client {  
    public static void main(String args[]) {  
        Pizza pizza = new TomatoPizza();  
        Pizza decoratedPizza = new MozzarellaPizza(pizza);  
  
        decoratedPizza.getCost();           // 5.5  
        decoratedPizza.getIngredients();    // TOMATO MOZZARELLA  
  
        decoratedPizza = new PepperoniPizza(decoratedPizza);  
        decoratedPizza.getCost();           // 6.5  
        decoratedPizza.getIngredients();    // TOMATO MOZZARELLA PEPPERONI  
    }  
}
```

Note

- Un approccio simile è usato nelle classi per l'input/output di Java: stream, writer, reader
- Oltre a modificare il comportamento di base è possibile anche aggiungere nuovi comportamenti
- Decorator è simile a Proxy, ma permette di comporre diversi comportamenti

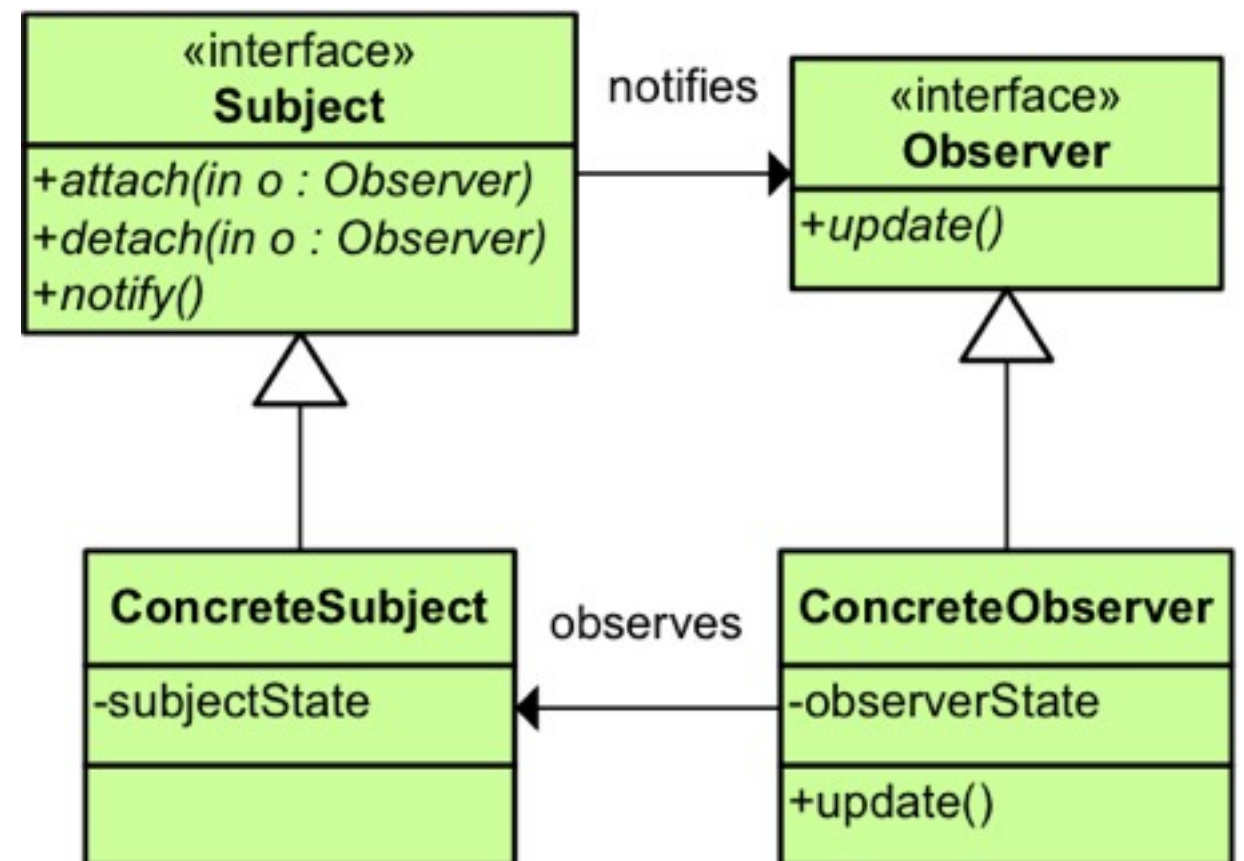
Pattern Comportamentali

Observer

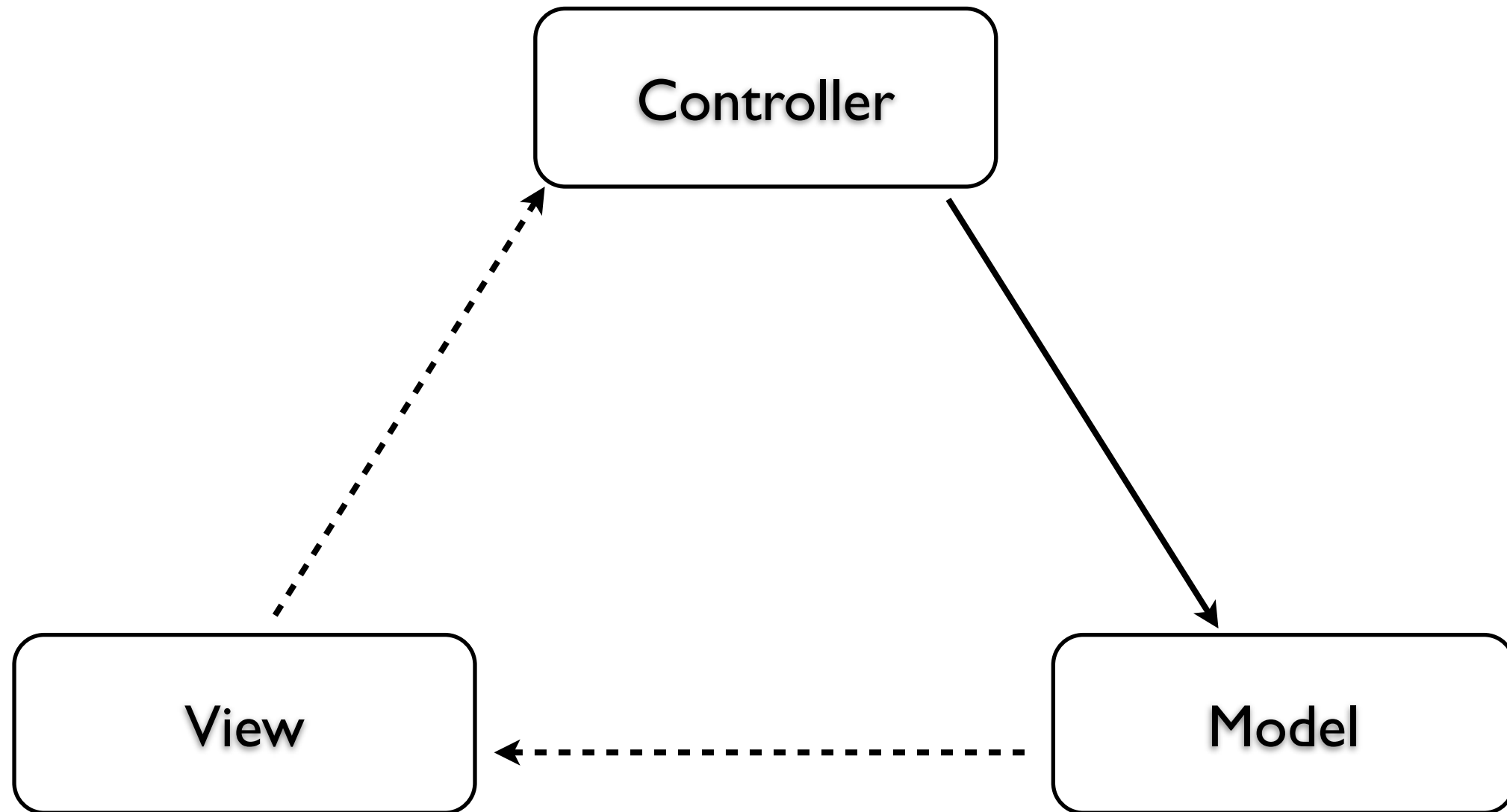
- Già visto in MVC
- Utile per gestire il paradigma ad eventi
- L'utilizzo che abbiamo visto riguarda la gestione degli eventi della GUI, ma c'è una versione più generale del pattern

Observer

- Utile per gestire il paradigma ad eventi
- Permette di gestire dinamicamente l'accoppiamento tra sorgenti di eventi e classi che devono reagire quando questi si verificano



Separation of concerns



Model

Incapsula lo stato dell'applicazione

Deve permettere di *accedere* ai dati

Deve *notificare* i cambiamenti dello stato

View

Deve *mostrare* il modello

Gestisce *l'interazione* con l'utente

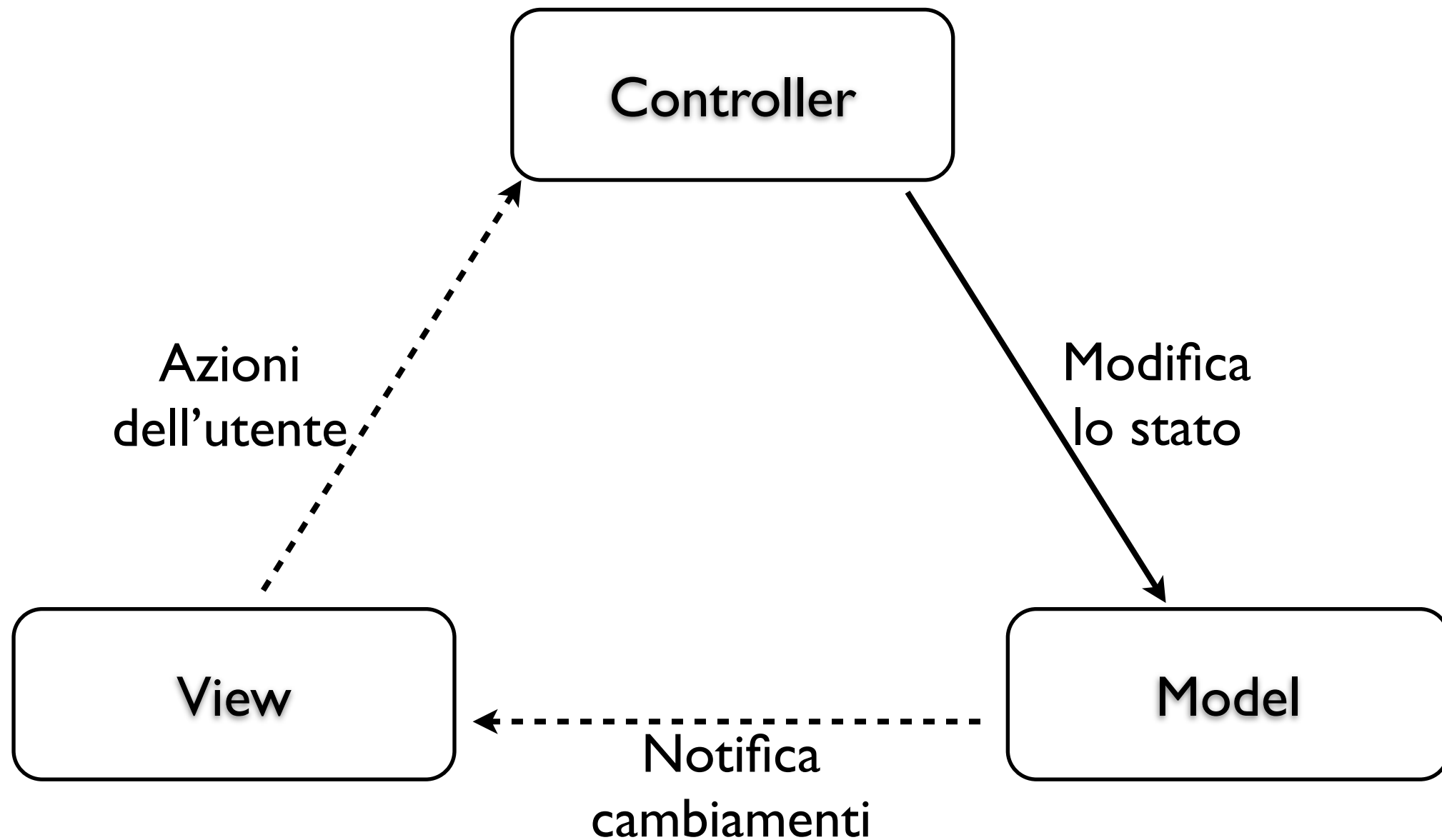
Controller

Rappresenta la *logica* applicativa

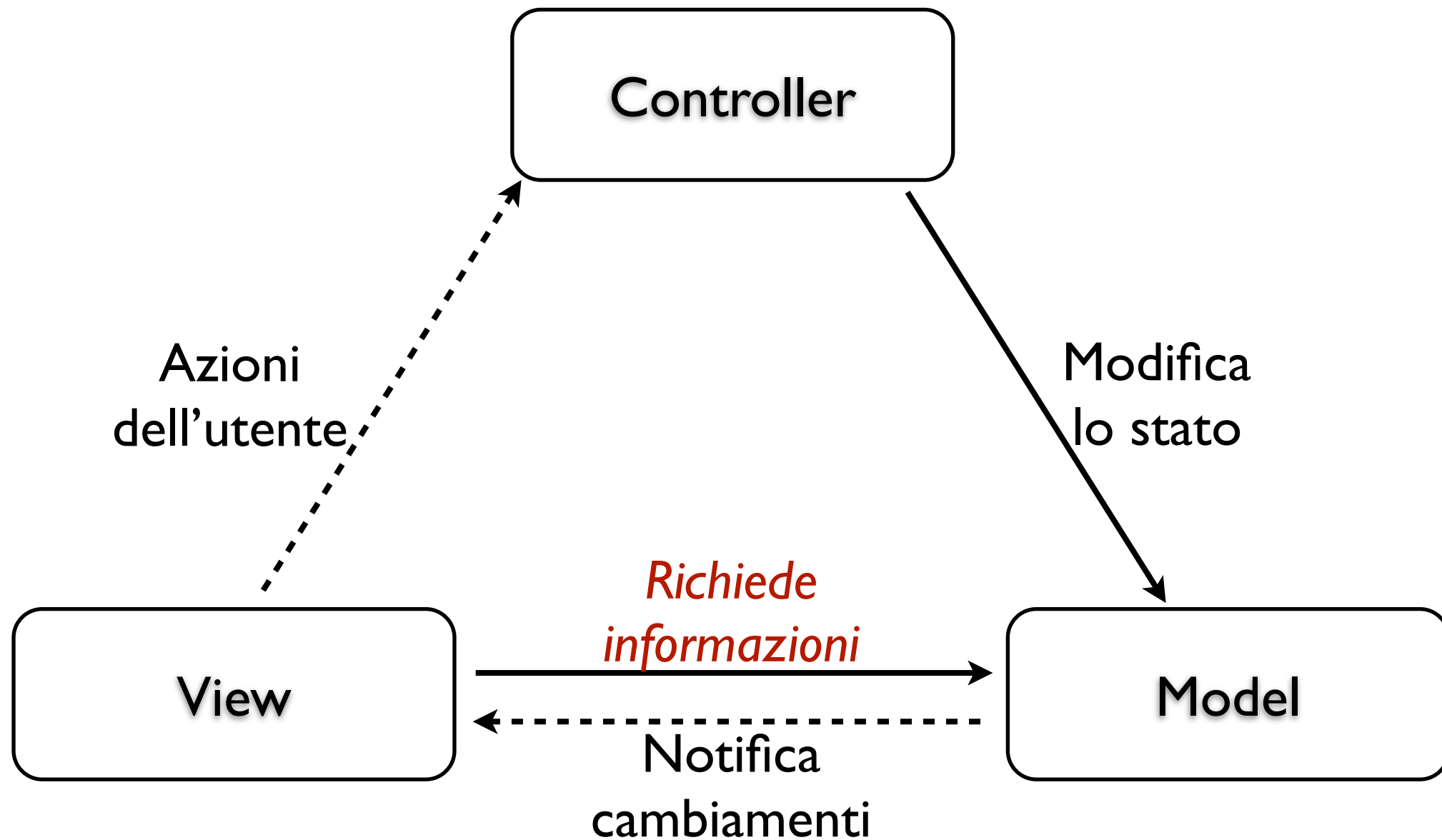
Collega le azioni dell'utente con modifiche allo stato

Sceglie cosa deve essere mostrato

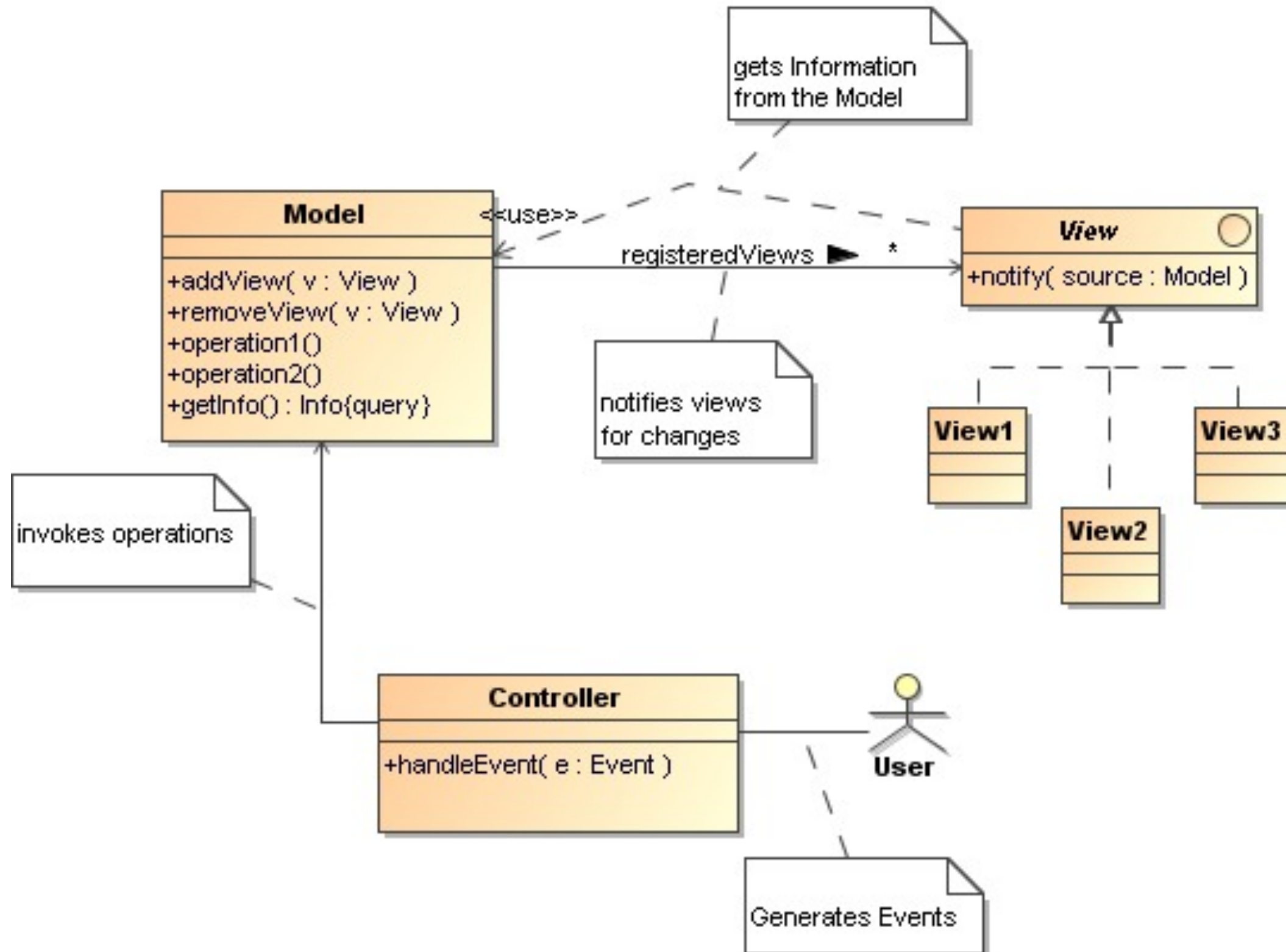
Come interagiscono?



Come interagiscono?

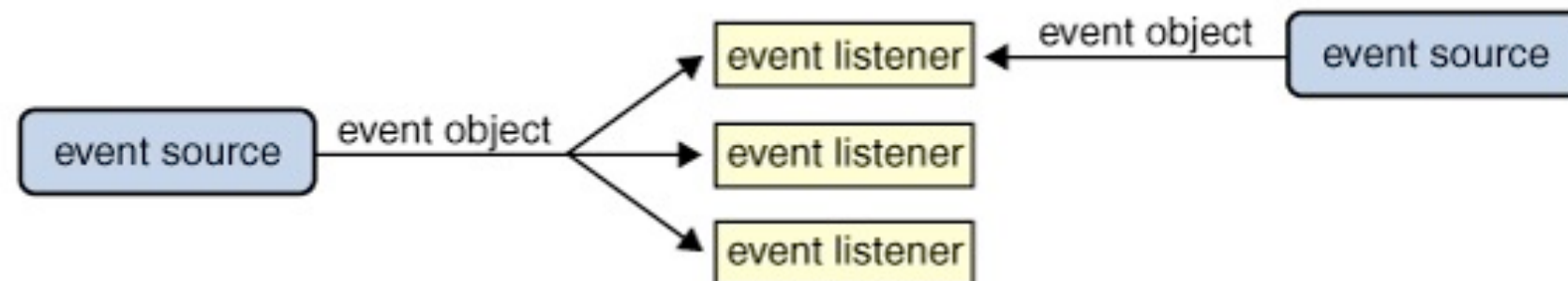


MVC: un possibile class diagram



Come si realizza?

Programmazione ad eventi per gestire la comunicazione tra i componenti:
Non solo tra View e Controller



Observer tra Model-View

Sorgenti di eventi

Notificano gli eventi agli interessati

La notifica avviene invocando i metodi sugli ascoltatori

Devono avere un **metodo** per permettere la registrazione degli ascoltatori

Esempio

```
public class EventSource{  
    private List<EventListener> listeners;  
  
    public void registerListener(EventListener listener){  
        listeners.add(listener);  
    }  
  
    private void notifyListeners(Event e){  
        for(EventListener listener : listeners){  
            listener.eventHappened(e);  
        }  
    }  
  
    public void foo() {  
        /* Viene generato un evento e inviata la notifica */  
        notifyListeners(new Event(...));  
    }  
}
```

Logica ed interfaccia

Logica ed interfaccia utente devono essere separate

Servono due interfacce:

Controller-Model/View

aggiornare la visualizzazione

View/Controller-Model

inviare i comandi e le richieste

Cosa passare

SEMPRE oggetti che rappresentano
eventi o oggetti specifici creati
appositamente

Oggetti modificabili del modello **NON**
devono arrivare all'interfaccia

Soluzioni

Classi di dialogo limitate (Solo metodi di visualizzazione)

Oggetti immutabili

Oggetti creati appositamente per la visualizzazione