UNIVERSITÀ DEGLI STUDI DI BRESCIA · unibs.it

AA 2026/27

PAJC · PROGRAMMAZIONE AVANZATA IN JAVA E C

L05

Pattern MVC e programmazione a eventi

Model, View e Controller sul codice del corso · il Model che avvisa: Observer, listener propri, `PropertyChangeSupport`



prof. Massimiliano Redolfi

Università degli Studi di Brescia · Ingegneria Informatica

Oggi

1. **Un giocattolo che non regge**: le palline del laboratorio e le tre domande
2. **Il Model**: il mondo come oggetti Java puri, provato senza finestra
3. **View e Controller**: chi disegna, chi comanda, chi misura il tempo
4. **Il triangolo MVC**: chi parla con chi, più View, MVC dentro Swing
5. **Il Model che avvisa**: Observer, listener propri, eventi come record
6. **Proprietà e regole**: `PropertyChangeSupport`, le regole, gli errori

01

Un giocattolo che non regge



Le palline del laboratorio

Un `JPanel` con un `javax.swing.Timer`: a ogni scatto sposta le palline e si ridisegna.

```
public class PnlPalline extends JPanel {
    private static class Pallina { double x, y, vx, vy, r; Color colore; }
    private final List<Pallina> palline = new ArrayList<>();
    private final Timer timer = new Timer(16, e -> passo());
    private void passo() {                          // velocita' in pixel per scatto
        for (Pallina p : palline) { p.x += p.vx; p.y += p.vy; /* rimbalzi sui bordi */ }
        repaint();
    }
    @Override protected void paintComponent(Graphics g) {
        /* ... le palline ... */
        g2.drawString(palline.size() + " palline", 10, getHeight() - 10);
    }
}
```

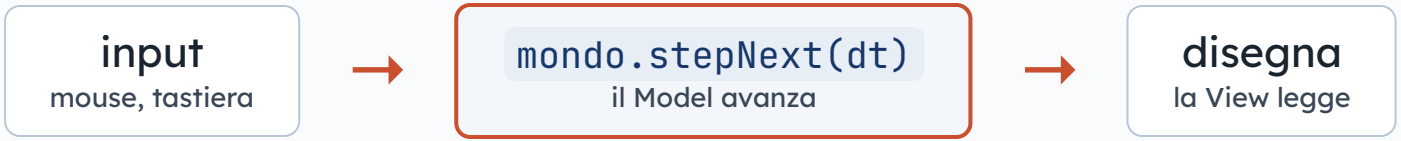
Funziona. Poi arrivano la gravità, il punteggio, un suono a ogni urto, la pausa, un secondo pannello.

Che cosa si rompe quando cresce

- **le velocità sono in pixel per scatto**: in una finestra grande le palline sono più lente, e se l'EDT è in ritardo il mondo rallenta con lui
- **lo stato vive nel pannello**: per sapere che cosa fa il mondo servono una finestra, un click e un occhio. Nessun test
- **la riga di stato è dentro** `paintComponent`: ogni cosa in più (punteggio, suono, pausa) finisce nella stessa classe, che cresce fino a mille righe
- **un secondo pannello sullo stesso mondo** non si può fare: le palline sono sue

> Le tre domande di ogni programma interattivo: **dove vive il mondo** e chi lo disegna; **come scorre il tempo**; **come si parlano le parti** senza conoscersi.

Il ciclo del gioco



Un mondo che non sa di Swing, al centro; attorno, un ciclo che lo fa avanzare e lo mostra

- il **Model** sta al centro e non conosce nessuno
- la **View** lo legge e non lo tocca
- il **Controller** lo tocca e non lo disegna

02

Il Model



Il mondo, senza la finestra

Che cos'è Palline, tolta la finestra? Una scatola con un pavimento, delle palline con posizione, velocità e raggio, una gravità, il tempo che passa. **Nessuna di queste cose è un pixel.**

- si misura in **metri e secondi**, origine in basso a sinistra, `y` verso l'alto: come in fisica
- posizioni e velocità sono **vettori**: un record immutabile, non `java.awt.Point`

```
public record Vec2(double x, double y) {
    public static final Vec2 ZERO = new Vec2(0, 0);
    public Vec2 piu(Vec2 v)   { return new Vec2(x + v.x, y + v.y); }
    public Vec2 per(double k) { return new Vec2(x * k, y * k); }
    public double modulo()    { return Math.hypot(x, y); }
}
```

La pallina

```
public class Pallina {
    private Vec2 posizione;                    // in metri, y in su
    private Vec2 velocita = Vec2.ZERO;
    private final double raggio;                 // fissato nel costruttore
    // posizione(), velocita(), raggio(): lo stato si legge da fuori
    void stepNext(double dt, Vec2 g) {         // prima la velocita', poi la posizione
        velocita = velocita.piu(g.per(dt));
        posizione = posizione.piu(velocita.per(dt));
    }
    void rimbalzaSulPavimento(double restituzione) {
        posizione = new Vec2(posizione.x(), raggio);
        velocita = new Vec2(velocita.x(), -velocita.y() * restituzione);
    }
}
```

Lo stato si legge da fuori; a muoverla è solo il mondo. Il rimbalzo è ingenuo, ma è **nel posto giusto**.

Il mondo

```
public class Mondo {
    public static final Vec2 G = new Vec2(0, -9.81);          // m/s^2
    private final List<Pallina> palline = new ArrayList<>();
    public List<Pallina> palline() { return Collections.unmodifiableList(palline); }
    public void lascia(Vec2 punto, double raggio) {           // da qui cade da sola
        palline.add(new Pallina(punto, raggio));
    }
    public void stepNext(double dt) {                         // dt in secondi
        for (Pallina p : palline) {
            p.stepNext(dt, G);
            if (p.posizione().y() < p.raggio()) p.rimbalzaSulPavimento(0.8);
        }
    }
}
```

- nomi del **dominio** (`lascia`, `stepNext`), non dell'interfaccia (`mousePressed`, `tick`)
- la lista esce **non modificabile**; nessun `import` da `java.awt` o `javax.swing`

Regola: niente AWT nel Model

> La tentazione: `Point` per la posizione, `Color` per il materiale, `Rectangle` per la scatola. Compila, funziona, e si paga dopo.

- `Point` ha coordinate **intere**: una pallina che si muove di 0,4 pixel per passo non si muove mai; `x` e `y` sono pubblici e mutabili
- un Model che importa AWT **non gira senza schermo**: non si prova senza finestra, non si mostra con un'altra libreria, non si manda in rete
- il colore non è della pallina ma del modo di disegnarla: la pallina ha un *materiale*, e la View decide che la gomma è arancione

```
Nel Model non si importa ne' java.awt ne' javax.swing.
```

Il premio: un mondo senza finestra

```
@Test
void unaPallinaLasciataCadeComeDiceLaFisica() {
    Mondo mondo = new Mondo(8, 6);
    mondo.lascia(new Vec2(4, 5.5), 0.25);
    for (int i = 0; i < 1000; i++)           // un secondo a passi di 1 ms
        mondo.stepNext(0.001);
    double y = mondo.palline().get(0).posizione().y();
    assertEquals(5.5 - 9.81 / 2, y, 0.01);   // y0 - g t^2 / 2
}
```

- nessuna finestra, nessun timer, nessun EDT: il mondo evolve da solo, e i numeri sono quelli della fisica; un decimo di secondo, da rieseguire a ogni modifica

> Il criterio più onesto: **se il test non si riesce a scrivere, il Model non è ancora separato**. È un pannello con un altro nome.

03

View e Controller



La View legge e disegna

```
private final Mondo mondo;                                  // ricevuto nel costruttore
private double scala() { /* pixel per metro: il mondo intero sta nel pannello */ }
private double worldToScreenX(double x) { return BORDO + x * scala(); }
private double worldToScreenY(double y) { return getHeight() - BORDO - y * scala(); }

@Override protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2 = (Graphics2D) g;
    for (Pallina p : mondo.palline()) {                     // legge, non scrive
        double r = p.raggio() * scala();
        g2.fill(new Ellipse2D.Double(worldToScreenX(p.posizione().x()) - r,
                                     worldToScreenY(p.posizione().y()) - r, 2 * r, 2 * r));
    }
}
```

Legge il mondo e lo traduce in pixel, **in quel momento**: non tiene copie, non chiama `stepNext`, non sa quanto tempo è passato.

`paintComponent` legge, non avanza

«Tanto viene chiamato a ogni fotogramma: facciamo avanzare il mondo lì.»

> `paintComponent` è un metodo che Swing chiama **quando vuole**: anche zero volte, anche cento.

- con la finestra coperta o ridotta a icona Swing non disegna: **il mondo si ferma**
- due `repaint()` ravvicinati vengono fusi in uno: **un passo perso**
- due View sullo stesso mondo: **il tempo passa due volte**
- e il test senza finestra non si scrive più

Per lo stesso motivo la View non tiene copie dei dati: li legge **nel momento in cui disegna**, e non può essere in ritardo.

Il Controller collega

```
public class Controller {
    private final Mondo mondo;  private final PnlMondo view;
    private final Timer timer = new Timer(16, e -> stepNext());
    private long ultimo;                                    // nanoTime() del passo prima
    public Controller(Mondo mondo, PnlMondo view) {
        this.mondo = mondo;  this.view = view;
        view.addMouseListener(new MouseAdapter() {
            @Override public void mousePressed(MouseEvent e) {
                mondo.lascia(view.screenToWorld(e.getPoint()), 0.25);
            }
        });
    }
    public void avvia() { ultimo = System.nanoTime(); timer.start(); }
}
```

- conosce **entrambi**: ascolta la View, scrive nel mondo
- possiede il **timer**, cioè il ciclo del gioco

Il periodo del timer non è il `dt`

```
private void stepNext() {                    // chiamato dal timer, sull'EDT
    long adesso = System.nanoTime();
    double dt = (adesso - ultimo) / 1e9;     // secondi passati davvero
    ultimo = adesso;
    mondo.stepNext(dt);
    view.repaint();
}
```

- `mondo.stepNext(0.016)` è l'errore dei pixel per scatto: 16 ms è una **richiesta**, l'evento parte quando l'EDT è libero
- su un computer carico i passi si diradano: a passo fisso **il tempo del mondo rallenta**; misurato, avanza di quanto deve

> `ultimo` si legge **quando il timer parte**, e a ogni ripartenza: altrimenti il primo `dt` vale ore.

Il `main` monta tutto

```
public static void main(String[] args) {
    SwingUtilities.invokeLater(() -> {
        Mondo mondo = new Mondo(8, 6);                          // metri
        PnlMondo view = new PnlMondo(mondo);
        Controller controller = new Controller(mondo, view);
        JFrame frame = new JFrame("Palline");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.add(view);
        frame.setSize(640, 480);
        frame.setVisible(true);
        controller.avvia();
    });
}
```

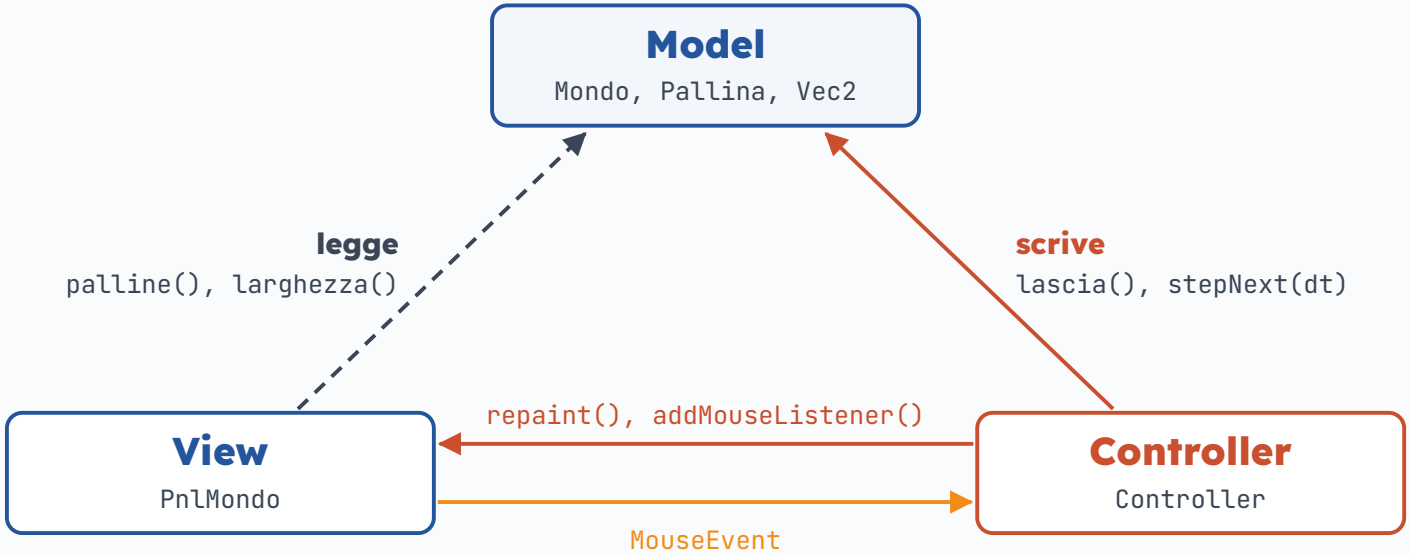
Prima il mondo, poi la View, poi il Controller; la finestra per ultima. Per ogni cosa in più c'è un posto: i rimbalzi nel mondo, i colori nella View, i tasti nel Controller.

04

Il triangolo MVC



Chi parla con chi



Le frecce permesse: il Controller scrive nel Model e comanda la View; la View legge il Model; il Model non conosce nessuno dei due

Le regole, con il perché

Chi	Che cosa fa	Che cosa non fa
Model	tiene lo stato del dominio e lo fa evolvere (`lascia`, `stepNext`); annuncia i cambiamenti (fra poco)	importare AWT o Swing; conoscere View e Controller; sapere che esiste un mouse
View	legge il Model e lo disegna; traduce fra metri e pixel; riceve l'input grezzo	modificare il Model; tenere copie dei suoi dati; decidere che cosa *significa* un click; far passare il tempo
Controller	crea e collega; traduce l'input in operazioni del Model; misura il tempo; chiede il ridisegno	disegnare; calcolare la fisica; tenere lo stato del dominio

> Il Controller è **l'unico che scrive**: ogni cambiamento del mondo ha un'origine sola, e quando qualcosa va storto c'è un posto solo in cui guardare.

Più View, un Model

- un secondo `PnlMondo` sullo stesso `Mondo` in un'altra finestra; una riga di stato con il numero di palline; una View testuale a console: **nessuna sa delle altre, il mondo non sa di nessuna**
- la View usa del mondo tre metodi di lettura: un'**interfaccia di sola lettura** al posto della classe, e il compilatore le impedisce di scrivere
- MVC non è «una classe per ruolo»: un gioco ha più View (il campo, il punteggio) e più terne per parti diverse

> Con due View e un solo Controller che fa `repaint()` sulla prima, la seconda resta ferma a mezz'aria finché Swing non decide di ridisegnarla. Si corregge nel Controller; il Model non cambia, perché non ha mai saputo quante View lo guardano.

MVC dentro Swing

Non è uno schema applicato a Swing da fuori: Swing stesso è fatto così, in piccolo, dentro ogni componente.

Componente	Model	Che cosa tiene
`JSlider`, `JProgressBar`	`BoundedRangeModel`	valore, minimo, massimo
`JComboBox`, `JList`	`ComboBoxModel`, `ListModel`	le voci, la selezione
`JTable`	`TableModel`	righe, colonne, celle
`JTextField`, `JTextArea`	`Document`	il testo
`JButton`, `JCheckBox`	`ButtonModel`	premuto, selezionato, abilitato

Uno `JSlider` non tiene il proprio valore: lo legge dal Model quando si disegna, e due slider sullo stesso Model si muovono insieme. Il componente è insieme View e Controller, e il Model **lo avvisa con un evento** quando cambia.

05

Il Model che avvisa



Chiedere o farsi avvisare

La riga di stato vuole «5 palline, 2 ferme»; il suono vuole un *toc* quando una pallina rimbalza.

- **polling**: il Controller, a ogni passo, conta e scrive il testo. Deve conoscere l'etichetta, e ogni View che si aggiunge; conta sessanta volte al secondo per un testo che cambia ogni tanto; e il *toc* non sa quando farlo partire
- **eventi**: chi **sa** avvisa. Il mondo è l'unico che sa quando una pallina viene lasciata, rimbalza, si ferma, perché è lui a farlo succedere: lo dice a chi si è dichiarato interessato

> Il mondo diventa una **stazione radio**: trasmette, e non sa se ascoltano un'etichetta, un suono, un registratore su file o nessuno. Chi scrive un listener nuovo non tocca il mondo.

Lo stesso pattern, in due forme

OBSERVER (GAMMA ET AL.)

- il **soggetto** offre `attach`, `detach`, e chiama `notify`
- l'**osservatore** ha un solo `update()`
- riceve «sono cambiato» e deve tornare dal soggetto a chiedere che cosa

DELEGA DEGLI EVENTI (JAVA)

- la **sorgente** offre `addXxxListener`, `removeXxxListener`
- il **listener** ha un metodo per tipo di fatto
- l'**evento** porta la sorgente e i dati del fatto: quale pulsante, quale pallina, a che velocità

L’evento e chi lo ascolta

```
public record Rimbalzo(Pallina pallina, double velocita) { }   // i dati del fatto

public interface MondoListener {
    default void pallinaLasciata(Pallina p) { }
    default void rimbalzo(Rimbalzo r)       { }
    default void pallinaFerma(Pallina p)    { }
}
```

- l’evento è un **record**: immutabile, con i dati e niente altro. La velocità d’impatto si prende *nel momento* dell’urto: un istante dopo è già quella di risalita
- i metodi hanno corpo vuoto con `default`: chi implementa ridefinisce **solo quelli che gli interessano**. Swing, nato prima, risolve lo stesso problema con gli *adapter* (`MouseAdapter`)

La sorgente: add, remove, fire

```
private final List<MondoListener> listeners = new ArrayList<>();
public void addMondoListener(MondoListener l)    { listeners.add(l); }
public void removeMondoListener(MondoListener l) { listeners.remove(l); }

public void lascia(Vec2 punto, double raggio) {
    Pallina p = new Pallina(punto, raggio);
    palline.add(p);                                    // prima nello stato
    for (MondoListener l : List.copyOf(listeners))     // poi l'annuncio, su una copia
        l.pallinaLasciata(p);
}
```

- la pallina è **aggiunta e solo dopo annunciata**: un listener che chiede `mondo.palline()` la trova
- la notifica scorre una **copia** (o `EventListenerList`): un listener che si cancella mentre viene avvisato cambierebbe la lista sotto il `for`

A passo concluso

```
public void stepNext(double dt) {
    List<Rimbalzo> rimbalzi = new ArrayList<>();
    for (Pallina p : palline) {
        p.stepNext(dt, G);
        if (p.posizione().y() < p.raggio()) {
            double impatto = -p.velocita().y();
            p.rimbalzaSulPavimento(0.8);
            rimbalzi.add(new Rimbalzo(p, impatto));          // raccolto, non annunciato
        }
    }
    for (MondoListener l : List.copyOf(listeners))           // a passo concluso
        for (Rimbalzo r : rimbalzi) l.rimbalzo(r);
}
```

I rimbalzi non si annunciano dove si scoprono, a metà del ciclo, quando metà delle palline sono avanzate e metà no: si **raccolgono** e si annunciano a stato coerente.

La View che ascolta

```
public class PnlStato extends JLabel implements MondoListener {
    private int lasciate, ferme;
    public PnlStato(Mondo mondo) {
        super("nessuna pallina");
        mondo.addMondoListener(this);
    }
    @Override public void pallinaLasciata(Pallina p) { lasciate++; aggiorna(); }
    @Override public void pallinaFerma(Pallina p)    { ferme++;    aggiorna(); }

    private void aggiorna() { setText(lasciate + " palline, " + ferme + " ferme"); }
}
```

- il Controller non la conosce, il mondo non la conosce: il `main` la crea e la mette in fondo alla finestra
- un `setText` per pallina lasciata o fermata, non sessanta al secondo

Ridisegnare a ogni passo, o su evento?

Perché `PnlMondo` non si registra sul mondo e fa `repaint()` a ogni evento?

- in un'animazione le palline si muovono a **ogni** passo: un evento «il mondo è avanzato» sessanta volte al secondo sarebbe solo un giro più lungo per arrivare allo stesso `repaint`
- il ridisegno **su evento** conviene quando il Model cambia **di rado e a scatti**: una scacchiera a ogni mossa, una lista a ogni inserimento, la riga di stato
- il ridisegno **a ogni passo** conviene quando il Model cambia in continuazione

> In un gioco coesistono: il pannello di gioco a ogni passo del ciclo, il punteggio e la riga di stato su evento.

La View che annuncia

Il Controller riceveva `mousePressed` e chiedeva `screenToWorld`: la View può parlare la lingua del mondo.

```
public interface PuntoListener { void puntoScelto(Vec2 punto); }
addMouseListener(new MouseAdapter() {                    // in PnlMondo
    @Override public void mousePressed(MouseEvent e) {
        Vec2 p = screenToWorld(e.getPoint());
        for (PuntoListener l : List.copyOf(listeners)) l.puntoScelto(p);
    }
});
```

```
view.addPuntoListener(p -> mondo.lascia(p, 0.25));     // il Controller, per intero
```

Il Controller decide solo che cosa il punto **significa**; le frecce del triangolo sono tutte letture o eventi.

06

Proprietà e regole



Fatti e proprietà

Un rimbalzo è un **fatto**: succede e passa. Gravità e pausa sono **proprietà**: ci sono sempre, e ogni tanto cambiano. Per loro Java ha `java.beans.PropertyChangeSupport`, che non dipende da Swing.

```
private double gravita = 9.81;
private final PropertyChangeSupport proprieta = new PropertyChangeSupport(this);

public void addPropertyChangeListener(PropertyChangeListener l) {
    proprieta.addPropertyChangeListener(l);             // remove allo stesso modo
}
public void setGravita(double g) {
    double vecchia = gravita;
    gravita = g;
    proprieta.firePropertyChange("gravita", vecchia, g);  // nome, vecchio, nuovo
}
```

Il listener è una lambda che riceve nome, vecchio e nuovo. `PropertyChangeSupport` **per i valori che cambiano; un'interfaccia propria e un record per i fatti che succedono.**

Nei due versi, senza girare in tondo

```
sldGravita.setValue((int) Math.round(mondo.gravita() * 100));            // centesimi
sldGravita.addChangeListener(e -> mondo.setGravita(sldGravita.getValue() / 100.0));
mondo.addPropertyChangeListener(e -> {
    if (e.getPropertyName().equals("gravita"))
        sldGravita.setValue((int) Math.round((Double) e.getNewValue() * 100));
});
```

- l'utente muove lo slider: il mondo cambia e annuncia, lo slider riceve il valore che ha già, e `setValue` con lo stesso valore **non fa scattare** il `ChangeListener`
- la gravità cambia altrove: lo slider si sposta e chiama `setGravita` con lo stesso valore, e `PropertyChangeSupport` **non notifica se vecchio e nuovo sono uguali**

> Regge se la conversione è **esatta nei due versi**: con lo slider in metri interi 9,81 diventerebbe 10 da solo; con due arrotondamenti diversi il giro non si chiude mai.

Le regole

1. **Si annuncia a stato coerente**: `lascia` aggiunge prima di annunciare, `stepNext` annuncia a passo concluso; mai dal costruttore
2. **L'evento è immutabile e porta i dati, non il Model**: un record, che si può tenere, mettere in lista, riprodurre
3. **Chi si cancella durante la notifica non rompe niente**: la copia della lista, o `EventListenerList`
4. **Chi non si cancella resta per sempre**: la finestra chiusa continua a contare palline nel vuoto e resta in memoria. Chi si registra si cancella
5. **Un solo thread**: timer, mouse, `stepNext` e notifiche girano sull'EDT. Un mondo su un thread suo rientrerebbe con `invokeLater`

Errori frequenti

- `Point` **e** `Color` **nel Model**: niente test, niente seconda View
- **la fisica in** `paintComponent`: il mondo si ferma con la finestra coperta
- `stepNext(0.016)`: il tempo del mondo rallenta con il computer
- `ultimo` **non riallineato alla ripartenza**: un `dt` di ore
- **la View che scrive nel Model** in `mousePressed`: due View litigano
- **annunciare a metà di un passo**: il listener legge uno stato incoerente
- **notificare scorrendo la lista originale**: `ConcurrentModificationException`
- **il listener dimenticato**: eventi nel vuoto e memoria che non si libera

MVC ed eventi, in due righe

- il **Model** è il dominio in oggetti Java puri, con `stepNext(dt)` in secondi e un test senza finestra; la **View** lo legge quando disegna; il **Controller** traduce l'input, misura il tempo e chiede il ridisegno
- il Model **annuncia** ciò che gli succede a chi si è registrato, senza conoscerlo: un record per il fatto, un'interfaccia con metodi `default`, la copia della lista; `PropertyChangeSupport` per le proprietà

> Per l'**esempio completo** del progetto Palline, e per ciò che viene dopo (fisica a passo fisso, materiali, comandi, stati, fabbriche): la dispensa «Pattern in gioco» e il materiale collegato.