Appunti di Progettazione del Software

Canale 1 · Prof. Paolo Liberatore

Capitolo 9: Oggetti reattivi e diagrammi degli stati

Autore: Tommaso C.

Appunti tratti dalle lezioni in aula e dal materiale del corso

---

Argomenti: Oggetti reattivi · Un thread e una coda per oggetto · Eventi come classi
Diagramma degli stati · Condizioni e azioni · Realizzazione in Java · Reattivo

Indice

1 Oggetti reattivi **1**

2 Eventi come classi **2**

3 Il diagramma degli stati e delle transizioni **2**

3.1 Azioni 3

3.2 Condizioni 3

4 Realizzazione in Java **4**

4.1 Gli eventi 4

4.2 Stati e ciclo degli eventi 5

5 Una superclasse per gli oggetti reattivi **7**

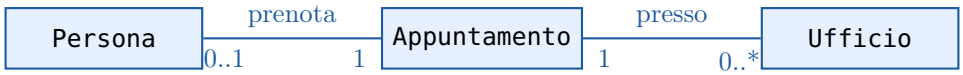
6 La realizzazione dei lucidi **8**

7 Esercizi **9**

8 Soluzioni **12**

1 Oggetti reattivi

Un sistema fissa appuntamenti fra persone e uffici. Lo schema dei dati ha tre classi: `Persona` (nome, cognome, anno di nascita), `Appuntamento` (data) e `Ufficio` (nome). Ogni appuntamento è prenotato da una persona presso un ufficio; una persona ha al più un appuntamento per volta, un ufficio quanti ne vuole.



Ogni persona segue un *processo* suo: è registrata senza richieste, aspetta una data, oppure ha un appuntamento fissato. Dieci persone possono stare in tre stati diversi, e ognuna avanza da sola; lo stesso vale per gli uffici.

Definizione – Oggetto reattivo

Un oggetto è **reattivo** se ha uno **stato** che cambia nel tempo e risponde ai messaggi che riceve; la risposta dipende dallo stato. Un oggetto che non riceve e non invia messaggi è un oggetto *dati*, come `Appuntamento`.

Proposizione – Un thread e una coda per oggetto

Ogni oggetto reattivo ha il suo thread e la sua coda bloccante dei messaggi ricevuti. Gli altri thread fanno `add` sulla coda; il thread dell'oggetto fa `take` e tratta un messaggio per volta, in ordine di arrivo.

Con quattro persone e tre uffici girano quindi sette thread, con sette code; gli appuntamenti non hanno né thread né coda. Se tutto girasse in un solo thread, una persona che aspetta bloccherebbe tutte le altre.

Osservazione – Una coda anche dove basterebbe un posto

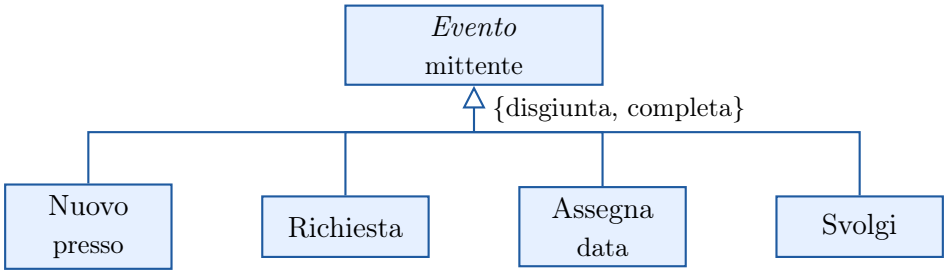
In questa prova una persona riceve un messaggio per volta, ma in generale ne può ricevere molti: per esempio chiede tre appuntamenti e aspetta tre risposte. Trattare tutti gli oggetti reattivi allo stesso modo è più semplice che distinguere i casi (Esercizio 1).

Nota – Dati, comportamento, interfaccia

I messaggi possono arrivare dal server web, dallo sportello o da un'altra parte del sistema. Per questo dati, comportamento e interfaccia restano separati: aggiungere un nuovo punto di accesso non obbliga a riscrivere il resto.

2 Eventi come classi

Nella coda di un oggetto si mettono messaggi di tipi diversi, quindi serve un tipo comune. Ogni messaggio è un oggetto di una sottoclasse di `Evento`; dalla classe si capisce che messaggio è, e i campi ne portano il contenuto.



- `Evento` è astratta: la generalizzazione è completa, ogni messaggio è di una delle sottoclassi.
- Ogni evento ha un **mittente**, che può essere `null` quando non conta: con più uffici, chi riceve una data vuole sapere quale ufficio l'ha mandata.
- Il contenuto cambia da evento a evento: `Nuovo` porta l'ufficio, `Assegna` la data, `Richiesta` e `Svolgi` solo il mittente.
- Gli eventi non si modificano dopo la creazione: hanno getter e non setter.

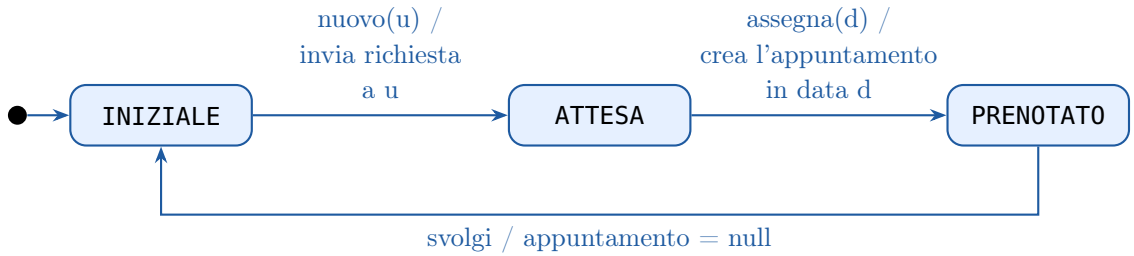
3 Il diagramma degli stati e delle transizioni

Il comportamento di una classe reattiva si disegna con un diagramma degli stati. Ogni stato è un rettangolo arrotondato; il pallino nero indica lo stato iniziale. Ogni freccia è una **transizione**: dice quale evento la scatena e che cosa succede.

Definizione – Etichetta di una transizione

evento`(`*parametri*`) [`*condizione*`] /` *azione*

La transizione scatta quando arriva l'evento e la condizione è vera. Poi l'oggetto esegue l'azione e passa allo stato di arrivo. Condizione e azione si possono omettere.



Il diagramma di `Persona` si legge così. Da `INIZIALE`, la richiesta di un appuntamento presso l'ufficio `u` invia a `u` una `Richiesta`. La persona aspetta allora in `ATTESA`; quando riceve `Assegna` con la data, crea l'appuntamento e lo collega. Dopo l'evento `svolgi`, torna in `INIZIALE` e può prenotare ancora.

Proposizione – Eventi ignorati

Un evento che non ha una transizione uscente dallo stato corrente non ha effetto: l'oggetto lo toglie dalla coda e resta nello stato in cui è.

L'Esercizio 2 mostra un varco che ignora l'apertura quando è già aperto.

3.1 Azioni

Un'azione può inviare un evento, più eventi o nessuno, e può modificare i dati dell'oggetto e dei suoi collegamenti. Un macchinario mostra i tre casi:

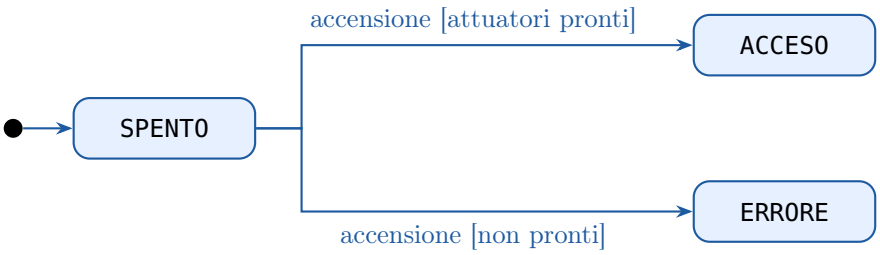
- `avvio` invia un evento a *ogni* operatore coinvolto;
- `rottura` invia un solo evento, al supervisore;
- `manutenzione` non invia nulla, ma aggiorna la data dell'ultima manutenzione.

Nota – Due passate

Conviene disegnare prima stati, eventi ricevuti ed eventi inviati, e solo dopo completare le azioni e le condizioni.

3.2 Condizioni

Uno stesso evento, nello stesso stato, può portare a stati diversi: le condizioni fra quadre scelgono la transizione. Per esempio un apparecchio spento, all'accensione, si accende se gli attuatori sono pronti e va in errore altrimenti.



Le due condizioni si escludono a vicenda: altrimenti il diagramma non direbbe quale transizione scegliere. I diagrammi sono fatti per essere letti da persone, quindi una condizione in parole va bene, purché sia chiara. Quando le condizioni cambiano solo l'azione e non lo stato di arrivo, basta una transizione (Esercizio 5).

Nota – Variabili di stato ausiliarie

Una transizione può dover ricordare un dato che non è un attributo della classe: per esempio un varco ricorda chi l'ha aperto, per avvisarlo se viene chiuso da altri. Questi dati sono **variabili di stato ausiliarie**, e la specifica le elenca accanto agli stati (Esercizi 3 e 4).

4 Realizzazione in Java

La realizzazione segue sempre lo stesso schema, che all'esame va usato così:

1. una classe astratta `Evento` con il mittente, e una sottoclasse per ogni evento, con il suo contenuto;
2. un'enumerazione degli stati e un campo `stato`, con lo stato iniziale;
3. una coda `ricevuti` e un thread per ogni oggetto reattivo;
4. un metodo `run` con un ciclo infinito: `take`, poi uno `switch` sullo stato e un controllo sulla classe dell'evento.

4.1 Gli eventi

```
public abstract class Evento {
    private Reattivo mittente;

    public Evento(Reattivo mittente) {
        this.mittente = mittente;
    }

    public Reattivo getMittente() {
        return mittente;
    }
```

```
    public String toString() {
        return getClass().getSimpleName();
    }
}
```

```
public class Nuovo extends Evento {
    private Ufficio presso;

    public Nuovo(Reattivo mittente, Ufficio presso) {
        super(mittente);
        this.presso = presso;
    }

    public Ufficio getPresso() {
        return presso;
    }
}
```

`Assegna` è uguale, con `String data` al posto dell'ufficio. `Richiesta` e `Svolgi` hanno solo il costruttore, che chiama `super(mittente)`.

4.2 Stati e ciclo degli eventi

```
public class Persona extends Reattivo {
    enum Stato { INIZIALE, ATTESA, PRENOTATO }

    private String nome;
    private String cognome;
    private int nascita;
    private Appuntamento appuntamento;
    private Stato stato = Stato.INIZIALE;

    public Persona(String nome, String cognome, int nascita) {
        this.nome = nome;
        this.cognome = cognome;
        this.nascita = nascita;
    }
```

```
    public void run() {
        while (true) {
            try {
                Evento e = ricevuti.take();
                switch (stato) {
                    case INIZIALE:
                        if (e.getClass() == Nuovo.class) {
                            Ufficio u = ((Nuovo) e).getPresso();
                            u.ricevuti.add(new Richiesta(this));
                            stato = Stato.ATTESA;
                        }
                        break;
                    case ATTESA:
                        if (e.getClass() == Assegna.class) {
                            Assegna a = (Assegna) e;
                            Ufficio u = (Ufficio) a.getMittente();
                            appuntamento =
                                new Appuntamento(a.getData(), this, u);
                            u.inserisci(appuntamento);
                            stato = Stato.PRENOTATO;
                        }
                        break;
                    case PRENOTATO:
                        if (e.getClass() == Svolgi.class) {
                            appuntamento = null;
                            stato = Stato.INIZIALE;
                        }
                        break;
                }
            } catch (InterruptedException ex) {
                return;
            }
        }
    }
}
```

- `e.getClass() == Nuovo.class` controlla che l'evento sia proprio di classe `Nuovo`.
- Il cast `(Nuovo) e` serve per leggere il contenuto: `getPresso` esiste in `Nuovo`, non in `Evento`.
- Ogni `case` controlla solo gli eventi che hanno una transizione da quello stato; gli altri passano senza effetto, come nel diagramma.
- Inviare un evento vuol dire aggiungerlo alla coda del destinatario.

Osservazione – Dati condivisi

`Persona` crea l'appuntamento nel suo thread, ma l'insieme degli appuntamenti sta nell'ufficio, che ha un thread suo. Per questo `inserisci` di `Ufficio` è un metodo `synchronized`.

L'ufficio ha un solo stato, quindi il suo `run` non ha bisogno dello `switch`:

```
public void run() {
    while (true) {
        try {
            Evento e = ricevuti.take();
            if (e.getClass() == Richiesta.class) {
                Persona p = (Persona) e.getMittente();
                p.ricevuti.add(new Assegna(this, "2026-11-20"));
            }
        } catch (InterruptedException ex) {
            return;
        }
    }
}
```

5 Una superclasse per gli oggetti reattivi

Persona e ufficio hanno in comune la coda, il thread e l'interfaccia `Runnable`. Queste parti vanno in una classe astratta, che crea la coda e avvia il thread nel costruttore:

```
public abstract class Reattivo implements Runnable {
    public BlockingQueue<Evento> ricevuti = new LinkedBlockingQueue<>();

    public Reattivo() {
        new Thread(this).start();
    }
}
```

La classe non definisce `run`: lo fa ogni sottoclasse, con il suo diagramma. Il thread creato con `new Thread(this)` esegue il `run` dell'oggetto stesso, perché l'oggetto è un `Runnable`.

Nota – Una coda pubblica

Solo il proprietario legge la coda, ma tutti gli altri devono poterci scrivere. Java non sa esprimere «privata per leggere, pubblica per scrivere», quindi la coda è `public`.

Osservazione – Il thread parte nel costruttore

Il thread parte prima che il costruttore della sottoclasse abbia assegnato i campi. Funziona perché `run` si ferma subito su `take`: un evento arriva solo quando l'oggetto è costruito e qualcuno ne conosce il riferimento.

6 La realizzazione dei lucidi

I lucidi del corso realizzano gli stessi diagrammi in un altro modo:

- ogni oggetto reattivo implementa un'interfaccia `Listener` con il metodo `Evento fired(Evento e)`, che esegue una transizione e restituisce l'evento da inviare, oppure `null`;
- un oggetto `Environment` tiene una coda per ogni listener e consegna gli eventi, uno per listener a ogni turno;
- ogni evento ha anche il destinatario, oltre al mittente.

Il corpo di `fired` è lo stesso `switch` della Sezione 4: per ogni stato, controlla la classe dell'evento e la condizione, esegue l'azione e cambia stato.

Nota – Due realizzazioni, un diagramma

Nella Sezione 4 la coda e il thread stanno nell'oggetto, e l'azione aggiunge l'evento alla coda del destinatario; nei lucidi l'`Environment` tiene le code e `fired` restituisce l'evento. Il diagramma degli stati non cambia.

Riepilogo

- **Oggetto reattivo**: ha uno stato e risponde ai messaggi in base allo stato; gli oggetti dati non ricevono né inviano messaggi.
- **Thread e coda**: ogni oggetto reattivo ha il suo thread e la sua `BlockingQueue`; inviare vuol dire fare `add` sulla coda del destinatario.
- **Eventi**: classe astratta `Evento` con il mittente, una sottoclasse per evento con il suo contenuto, senza setter.
- **Transizione**: *evento*`(par)[cond]/`*azione*; scatta se arriva l'evento e la condizione è vera.
- **Azioni e condizioni**: un'azione invia zero, uno o più eventi e cambia dati; condizioni esclusive scelgono fra transizioni con lo stesso evento.
- **Eventi ignorati**: senza transizione uscente, l'evento non ha effetto.
- **Java**: `enum` degli stati, `run` con `take`, `switch` sullo stato, `getClass()` e cast per il contenuto.
- **Reattivo**: superclasse astratta con coda pubblica e thread avviato nel costruttore; i lucidi usano `Listener`, `fired` ed `Environment`.

7 Esercizi

Esercizio 1 – Perché una coda per ogni persona

Nel sistema degli appuntamenti una persona ha al più un appuntamento, quindi riceve un messaggio per volta. Perché anche `Persona` ha una coda bloccante, e non una sola variabile per l'ultimo evento?

Soluzione a pagina 12 →

Esercizio 2 – Il varco

Un varco è inizialmente chiuso. L'evento `apri`, inviato da una persona, lo apre, e il varco ricorda chi l'ha aperto. Da aperto, l'evento `chiudi` lo chiude; anche l'evento `chiusuraGenerale` lo chiude, e in questo caso il varco invia l'evento `chiuso` alla persona che l'aveva aperto. Disegnare il diagramma degli stati, indicare le variabili ausiliarie e scrivere il `run` di `Varco extends Reattivo`.

Soluzione a pagina 12 →

Esercizio 3 – Appello del 26/06/2026, Canale 1, Prof. Paolo Liberatore, esercizio 1, adattato

Un'applicazione controlla i nidi delle tartarughe marine con dei rilevatori montati sulla spiaggia. Quando un rilevatore riprende una tartaruga che scava, crea un `Nido` con data e orario (stringhe); in seguito un `Segnalatore` arriva e segna il nido.

Il rilevatore è inizialmente *sospeso*. Un evento di *attivazione* da parte di un segnalatore lo porta in *attivo*, in cui riprende la spiaggia. Una deposizione causa l'evento *nuovo*, che lo porta in *presenza*; un'ulteriore deposizione lo lascia in presenza. Alla ricezione di nuovo il rilevatore crea un `Nido` con data e ora attuali. Un evento *segnato*, inviato dal segnalatore, collega il nido al segnalatore; quando non restano nidi non segnati, il rilevatore torna in attivo. Solo da attivo il rilevatore può sospendersi, con l'evento *fine*, e allora invia una *notifica* al segnalatore che l'aveva attivato l'ultima volta.

a) Disegnare il diagramma degli stati di `Rilevatore` e indicarne le variabili di stato ausiliarie.
b) Scrivere il `run` di `Rilevatore extends Reattivo`.

Soluzione a pagina 13 →

Esercizio 4 – Compito d'esame «Visite mediche», Prof. Paolo Liberatore, domanda 1, adattato

Un reparto ospedaliero gestisce le visite prenotate dai pazienti. Inizialmente una visita è *Prenotata*. Un evento *Assegna* la porta nello stato *Assegnata* e le assegna il medico specificato nell'evento. In condizioni normali, la visita diventa poi *Effettuata* a seguito di un evento *Effettua*. In caso di problemi, la visita può venire *Sospesa* da un evento *Sospendi*, che cancella l'assegnamento del medico. Una visita sospesa torna *Assegnata* a seguito di un evento *Riassegna*, che porta il nuovo medico; se questo è diverso dal medico a cui la visita era stata assegnata inizialmente, a quest'ultimo si invia una *Notifica* che include la visita. Una visita assegnata o sospesa diventa *Annullata* se riceve un evento *Annulla*.

a) Disegnare il diagramma degli stati di `Visita` e indicarne le variabili di stato ausiliarie.
b) Scrivere il `case SOSPESA` del `run` di `Visita extends Reattivo`.

Soluzione a pagina 15 →

Esercizio 5 – Appello del 20/02/2026, Canale 1, Prof. Paolo Liberatore, esercizio 1, adattato

In una banca, un appuntamento si svolge fra un consulente e da uno a cinque clienti; ogni cliente ha esattamente un consulente. Il consulente dell'appuntamento può essere diverso da quello dei clienti, e può cambiare.

Gli appuntamenti iniziano in stato *Prenotato*. In questo stato il consulente può cambiare a seguito di un evento *Cambia*, anche più volte. L'appuntamento inizia a seguito di un evento *Comincia*, che lo porta nello stato *Iniziato*. Se il consulente in quel momento è diverso da quello con cui l'appuntamento era stato creato, vengono informati girando a entrambi l'evento di inizio. Se il consulente in quel momento è diverso dal consulente dei clienti, il messaggio viene girato anche a quest'ultimo. Un appuntamento iniziato viene concluso da un messaggio *Finito*; un appuntamento prenotato può venire annullato da un messaggio *Cancellato*. Disegnare il diagramma degli stati di `Appuntamento` e indicarne le variabili di stato ausiliarie.

Soluzione a pagina 16 →

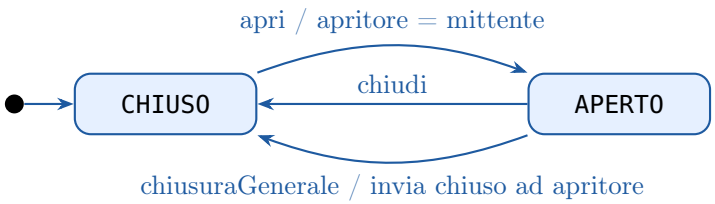
8 Soluzioni

Soluzione dell'Esercizio 1 – Perché una coda per ogni persona

Il thread della persona deve fermarsi finché non arriva un evento. Con una variabile, l'attesa e il risveglio andrebbero scritti a mano con `wait` e `notify`; la coda bloccante li fa già con `take`. In generale, poi, una persona può ricevere più messaggi insieme, per esempio le risposte di più uffici. Trattare tutti gli oggetti reattivi allo stesso modo evita due realizzazioni diverse.

← *Torna all'Esercizio 1, pagina 9*

Soluzione dell'Esercizio 2 – Il varco



Variabile ausiliaria: `apritore`, la persona che ha aperto il varco. Un `apri` su un varco aperto e un `chiudi` su un varco chiuso non hanno effetto.

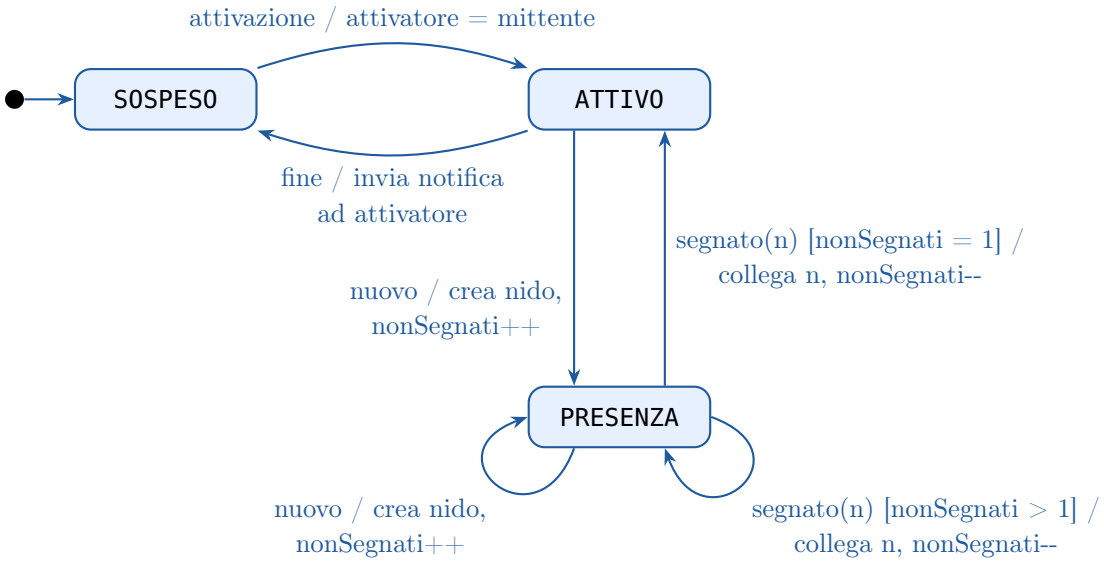
```
public void run() {
    while (true) {
        try {
            Evento e = ricevuti.take();
            switch (stato) {
                case CHIUSO:
                    if (e.getClass() == Apri.class) {
                        apritore = (Persona) e.getMittente();
                        stato = Stato.APERTO;
                    }
                    break;
                case APERTO:
                    if (e.getClass() == Chiudi.class) {
                        apritore = null;
                        stato = Stato.CHIUSO;
                    } else if (e.getClass() == ChiusuraGenerale.class) {
                        apritore.ricevuti.add(new Chiuso(this));
                        apritore = null;
                        stato = Stato.CHIUSO;
                    }
                    break;
```

```
            }
        } catch (InterruptedException ex) {
            return;
        }
    }
}
```

I campi sono l'enumerazione `Stato`, con `CHIUSO` e `APERTO`; il campo `stato`, che parte da `CHIUSO`; il campo `apritore`, di tipo `Persona`.

← *Torna all'Esercizio 2, pagina 9*

Soluzione dell'Esercizio 3 – Appello del 26/06/2026, Canale 1, Prof. Paolo Liberatore, esercizio 1, adattato



Variabili ausiliarie: `attivatore`, il segnalatore dell'ultima attivazione, e `nonSegnati`, il numero di nidi non ancora segnati. Un `fine` ricevuto in presenza non ha effetto.

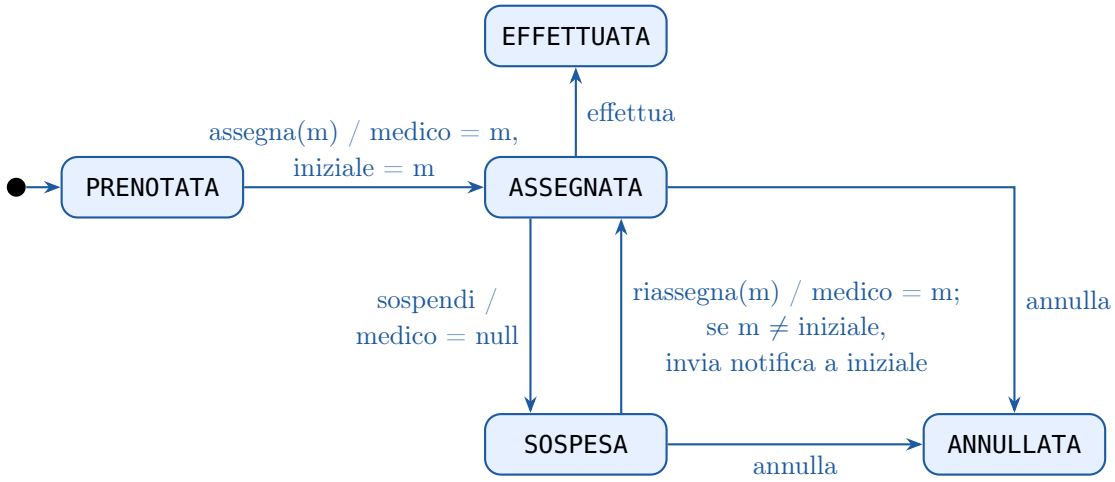
```
public void run() {
    while (true) {
        try {
            Evento e = ricevuti.take();
            switch (stato) {
                case SOSPESO:
                    if (e.getClass() == Attivazione.class) {
                        attivatore = (Segnalatore) e.getMittente();
                        stato = Stato.ATTIVO;
                    }
```

```
                    break;
                case ATTIVO:
                    if (e.getClass() == Nuovo.class) {
                        creaNido();
                        stato = Stato.PRESENZA;
                    } else if (e.getClass() == Fine.class) {
                        attivatore.ricevuti.add(new Notifica(this));
                        stato = Stato.SOSPESO;
                    }
                    break;
                case PRESENZA:
                    if (e.getClass() == Nuovo.class) {
                        creaNido();
                    } else if (e.getClass() == Segnato.class) {
                        Nido n = ((Segnato) e).getNido();
                        n.segnalatore = (Segnalatore) e.getMittente();
                        nonSegnati--;
                        if (nonSegnati == 0)
                            stato = Stato.ATTIVO;
                    }
                    break;
            }
        } catch (InterruptedException ex) {
            return;
        }
    }
}
```

`creaNido()` crea un `Nido` con `LocalDate.now()` e `LocalTime.now()` convertiti in stringa e incrementa `nonSegnati`. Le due transizioni di `segnato` diventano un solo ramo, con un `if` sul contatore dopo il decremento.

← *Torna all'Esercizio 3, pagina 10*

Soluzione dell'Esercizio 4 – Compito d'esame «Visite mediche», Prof. Paolo Liberatore, domanda 1, adattato



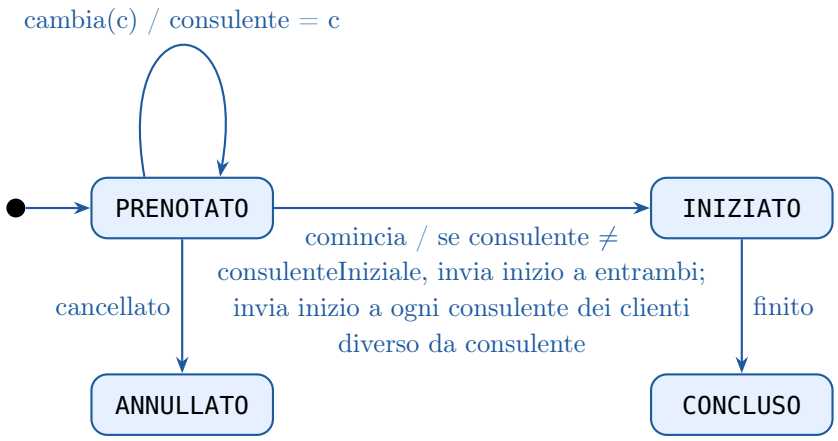
Variabile ausiliaria: `iniziale`, il medico della prima assegnazione. La assegna solo `assegna`: `riassegna` cambia il medico, ma non `iniziale`. Il medico attuale è il collegamento della visita, non una variabile ausiliaria.

```
case SOSPESA:
    if (e.getClass() == Riassegna.class) {
        medico = ((Riassegna) e).getMedico();
        if (medico != iniziale)
            iniziale.ricevuti.add(new Notifica(this));
        stato = Stato.ASSEGNATA;
    } else if (e.getClass() == Annulla.class) {
        stato = Stato.ANNULLATA;
    }
    break;
```

Il confronto usa `!=` perché conta se è lo stesso oggetto medico, non se due medici hanno gli stessi dati.

← *Torna all'Esercizio 4, pagina 10*

Soluzione dell'Esercizio 5 – Appello del 20/02/2026, Canale 1, Prof. Paolo Liberatore, esercizio 1, adattato



Variabile ausiliaria: `consulenteIniziale`, il consulente con cui l'appuntamento è stato creato; il consulente attuale è già il collegamento dell'appuntamento. Gli stati `ANNULLATO` e `CONCLUSO` non hanno transizioni uscenti: ogni evento successivo viene ignorato.

Le azioni di `comincia` dipendono da condizioni, ma lo stato di arrivo è sempre `INIZIATO`. Basta quindi una transizione con l'azione condizionata. Due transizioni con condizioni diverse servirebbero solo se portassero a stati diversi.

← *Torna all'Esercizio 5, pagina 11*