

Fondamenti di Informatica T2

Lab06 – Calendario appuntamenti

Corso di Laurea in Ingegneria Informatica

Anno accademico 2021/2022

Prof. ROBERTA CALEGARI

Prof. AMBRA MOLESINI

Dipartimento di Informatica – Scienza e Ingegneria (DISI)



Calendario Appuntamenti

- **OBIETTIVO: realizzare una applicazione per la gestione di un calendario di appuntamenti**
- L'applicazione dovrà offrire le seguenti funzionalità:
 - **Inserire** un nuovo appuntamento
 - **Cancellare** un appuntamento
 - **Mostrare** tutti gli appuntamenti
 - **Ottenere** la lista degli appuntamenti:
 - Giornaliera
 - Mensile
- **Ma... cos'è esattamente un *APPUNTAMENTO* ?**



Analisi del problema (1)

- **APPUNTAMENTO**

- descrizione testuale
- data/ora iniziale, data/ora finale
- durata

- **Quale concetto di "data/ora"?**

- in un calendario, il riferimento è *relativo all'utente*:
non è un sistema di gestione di voli aerei!
- **sono tutte date&ore LOCALI → LocalDateTime**

- **Quale concetto di "durata"?**

- **un appuntamento ha una durata precisa** (ore, minuti),
non solo una vaga nozione di "periodo" non meglio precisato
- **durata precisa → Duration**



Analisi del problema (2)

- **CALENDARIO**

- un calendario *gestisce una "collezione" di appuntamenti*
- infatti, deve permettere di
 - inserire / cancellare un appuntamento
 - ottenere l'elenco di tutti gli appuntamenti

- **Ok, ma... cos'è una *Collezione di appuntamenti* ?**



Analisi del problema (3)

- **COLLEZIONE DI APPUNTAMENTI**

- **sequenza di appuntamenti**
- *dimensione fisica **espandibile al bisogno***
- *dimensione logica **variabile***

- **Funzionalità**

- aggiunta / rimozione di un appuntamento
- ricerca di un appuntamento
 - per posizione (restituisce l'appuntamento i-esimo)
 - per appuntamento (restituisce la sua posizione nella sequenza)
[MA.. cosa significa esattamente "*ricerca per appuntamento*"?]

Organizzazione dell'applicazione

- L'applicazione è chiaramente *complicata*: non può essere strutturata come singolo componente
 - sarebbero *troppe funzionalità* in un singolo componente monolitico
 - non sarebbe gestibile, né riutilizzabile
- Inoltre, l'applicazione presenta *aspetti chiaramente disgiunti*
 - user interface (UI)
 - organizzazione interna (modello dei dati)
 - coordinamento fra le due
 - ...

La struttura è essenziale!
Spesso, più della parte
algoritmica..

Come organizzare
"bene" un sistema
come questo ?



Organizzazione dell'applicazione

- Esistono *Design Pattern* per queste situazioni, che garantiscono
 - strutture efficaci, chiare e ben manutenibili
 - massimo *disaccoppiamento* fra le parti (e quindi massima *riusabilità*)
 - massima *testability* (necessità di poterle *collaudare separatamente*)
- Elementi fondamentali in gioco
 - i dati gestiti → *modello dei dati*
 - la user interface → *una o più viste grafiche*
 - il coordinamento fra le due → il "*burattinaio*"
- Approcci tipici: MVC (, MVVM, MVP)
 - differiscono per lo specifico ruolo del "burattinaio" e lo schema preciso con cui interagisce con gli altri

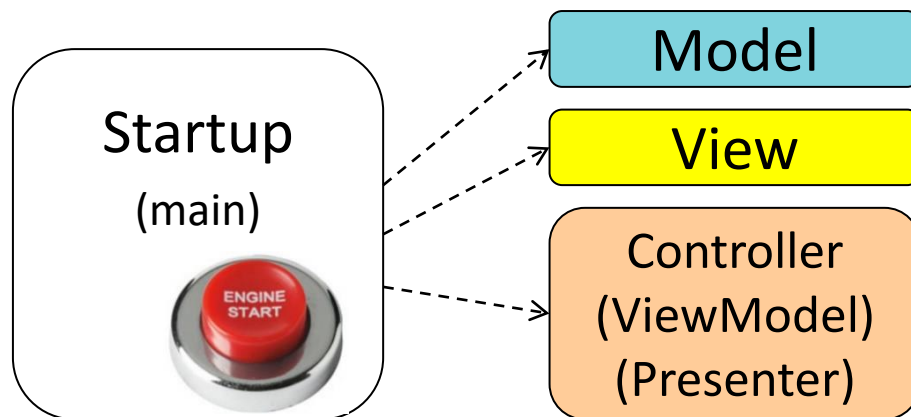
Model

View

Controller
(ViewModel)
(Presenter)

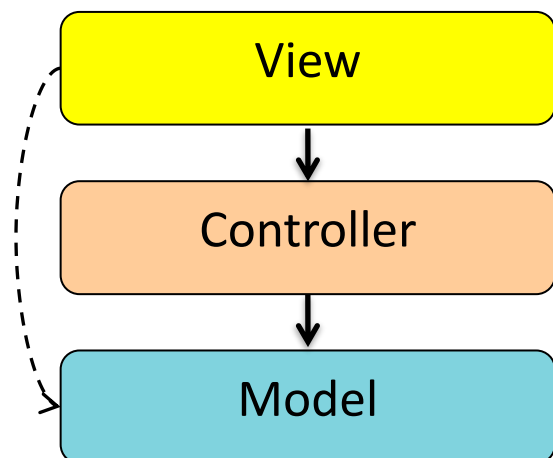
Organizzazione dell'applicazione

- Importante: **non preoccupiamoci di "dove sia" il main**
 - Model, View e Controller costituiscono il "core" dell'applicazione
 - il "main" non è nessuno di questi elementi
- Lo scopo del main (e della classe che lo ospita) è infatti **gestire tutta la fase di startup**, ossia:
 - *creare la struttura*
 - *connettere le parti*
 - *far partire l'applicazione*



Il pattern MVC

- **Model-View-Controller (MVC)** è il più classico di questi schemi
 - **Model:** costituisce l'insieme dei dati (oggetti) su cui l'applicazione opera
 - **View:** cura la rappresentazione visuale dei dati e l'interazione con l'utente
 - **Controller:** è il "burattinaio" che governa le elaborazioni sui dati
("non si muove foglia che il controller non voglia")

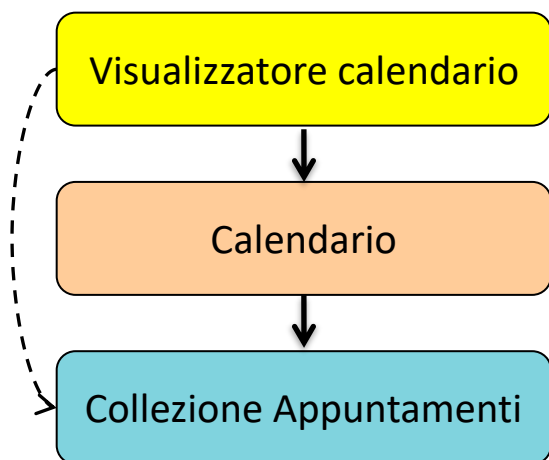


ASSIOMA: il flusso di controllo deve passare sempre dal **Controller**

*..anche se la **View** spesso deve avere una conoscenza del **Model** per poter visualizzare al meglio i dati (freccia tratteggiata)*

Calendario appuntamenti con MVC

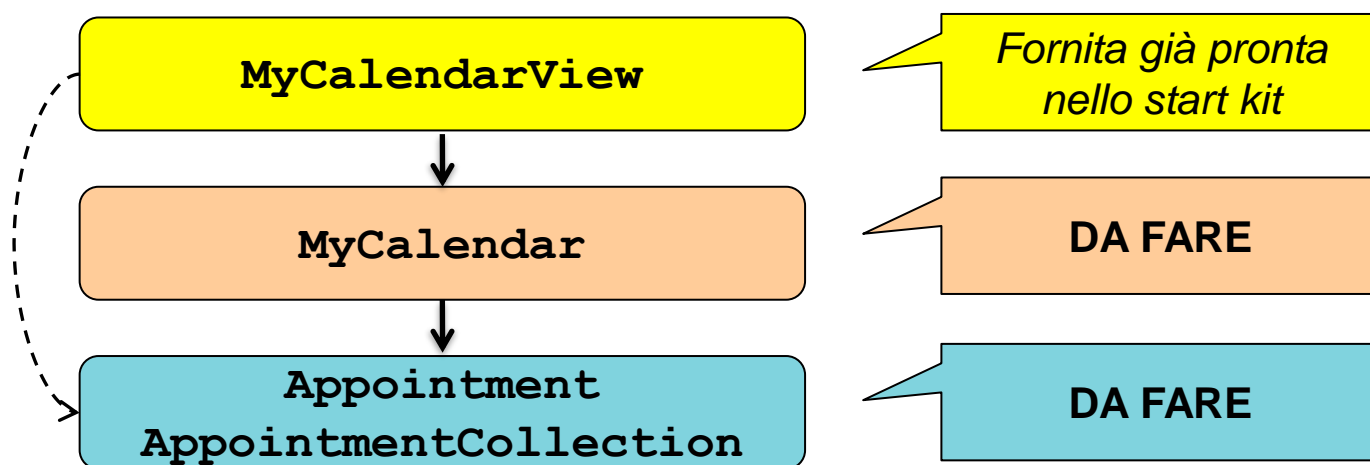
- Nel caso della nostra applicazione Calendario appuntamenti:
 - **Model:** gestisce la collezione di appuntamenti
 - **View:** interagisce con l'utente e visualizza i dati a console
 - **Controller:** elabora le richieste dell'utente = manipola/filtra appuntamenti



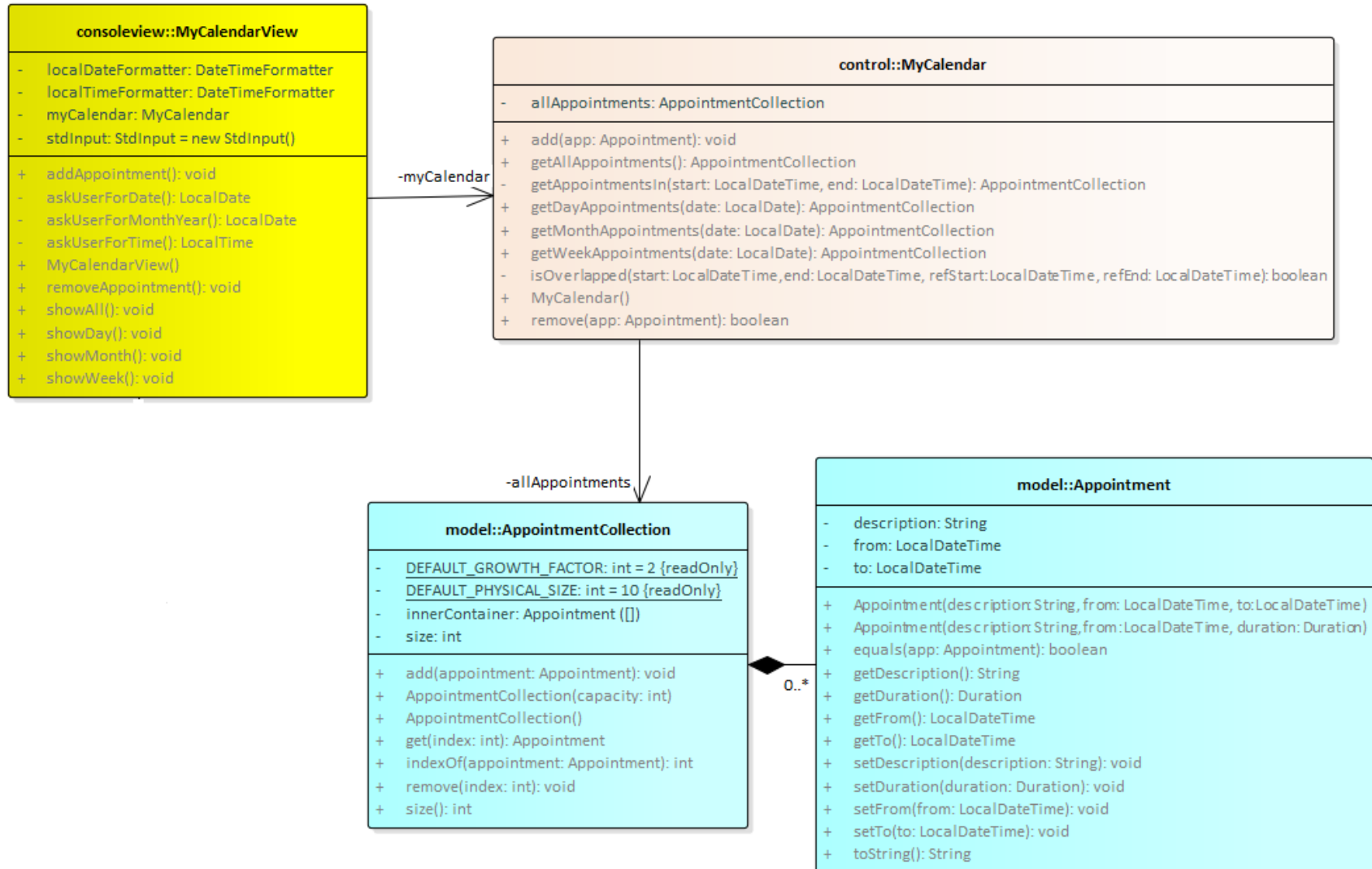
- Il **visualizzatore** *non opera mai* direttamente su (la collezione di) **appuntamenti**
 - anche se ne ha una certa conoscenza per poterli visualizzare al meglio
- il controllo passa sempre da **Calendario**
 - solo lui manipola gli **appuntamenti**

Calendario appuntamenti con MVC

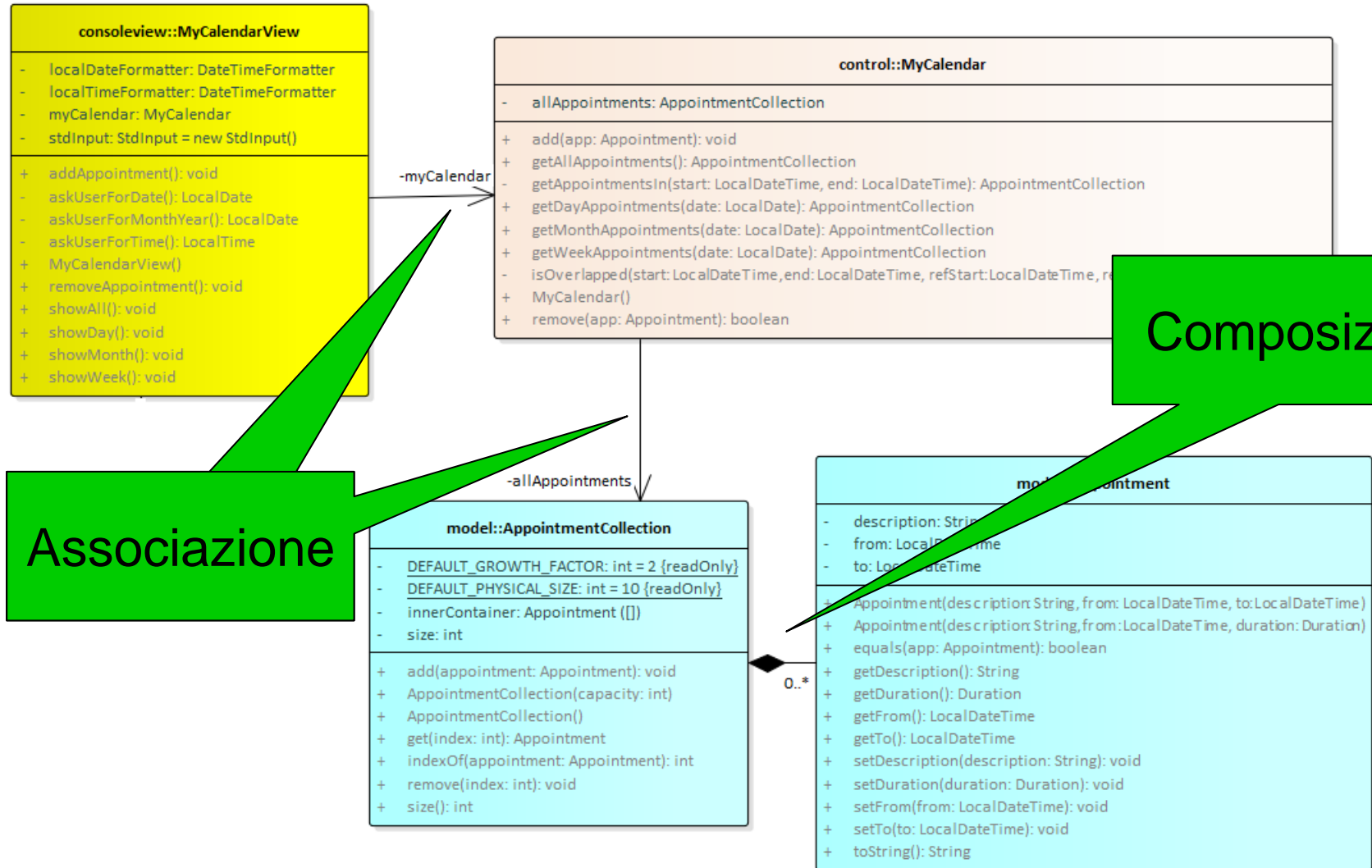
- Nel caso della nostra applicazione Calendario appuntamenti:
 - **Model:** due classi, **Appointment** e **AppointmentCollection**
 - **View:** classe **MyCalendarView** (*fornita già pronta nello start kit*)
 - **Controller:** classe **MyCalendar**



Organizzazione dell'applicazione



Organizzazione dell'applicazione



UML - Associazione

Associazione



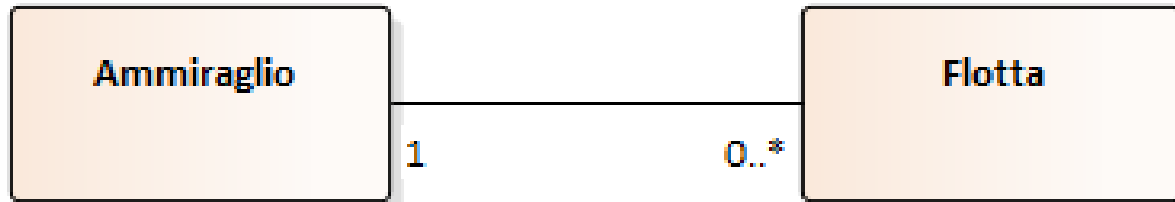
Particolare relazione tra due classi
che esprime un legame «forte»



Associazione

- generica (senza frecce) ogni classe contiene un riferimento all'altra
 - se molteplicità >1 , il riferimento sarà a un array o collection
- orientata (con frecce) solo la classe da cui parte la freccia ha un riferimento alla classe verso cui punta la freccia (come sopra il tipo di riferimento dipende dalla molteplicità)

Associazione Generica



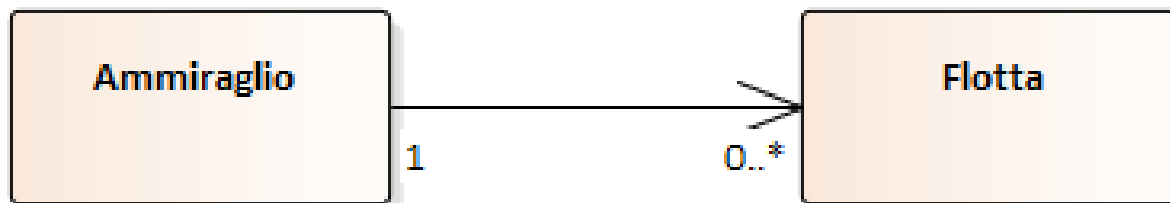
```
public class Ammiraglio {  
    private Flotta[] flotte ;  
}
```

```
public class Flotta {  
    private Ammiraglio amm;  
}
```

Non si specifica la «navigabilità» dell'associazione, di solito viene interpretato come associazione (navigabilità) bidirezionale

→ da Ammiraglio si può accedere a Flotta e viceversa

Associazione Orientata



```
public class Ammiraglio {  
    private Flotta[] flotte ;  
}
```

```
public class Flotta {  
}
```

Specifichiamo che l'associazione è «navigabile» solo in una direzione ben determinata

→ da Ammiraglio si può accedere a Flotta ma NON viceversa

UML - Composizione

Composizione



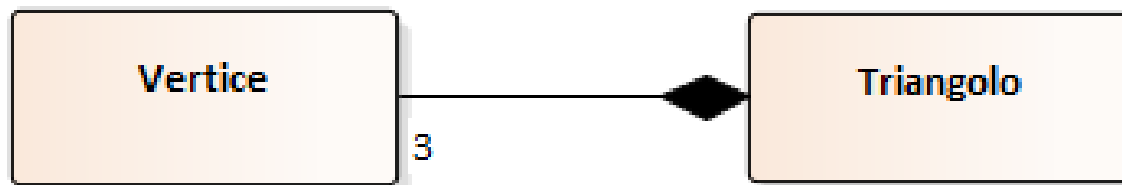
Particolare relazione tra due classi che esprime un legame forte «**intero-parte**»



Composizione: la classe «aggregante» contiene un riferimento a un array (o collection) dell'altra

- il tempo di vita delle classi aggregate è dipendente da quello della classe aggregante
 - è un contenimento esclusivo
 - è necessario fare una copia privata nel costruttore

Composizione

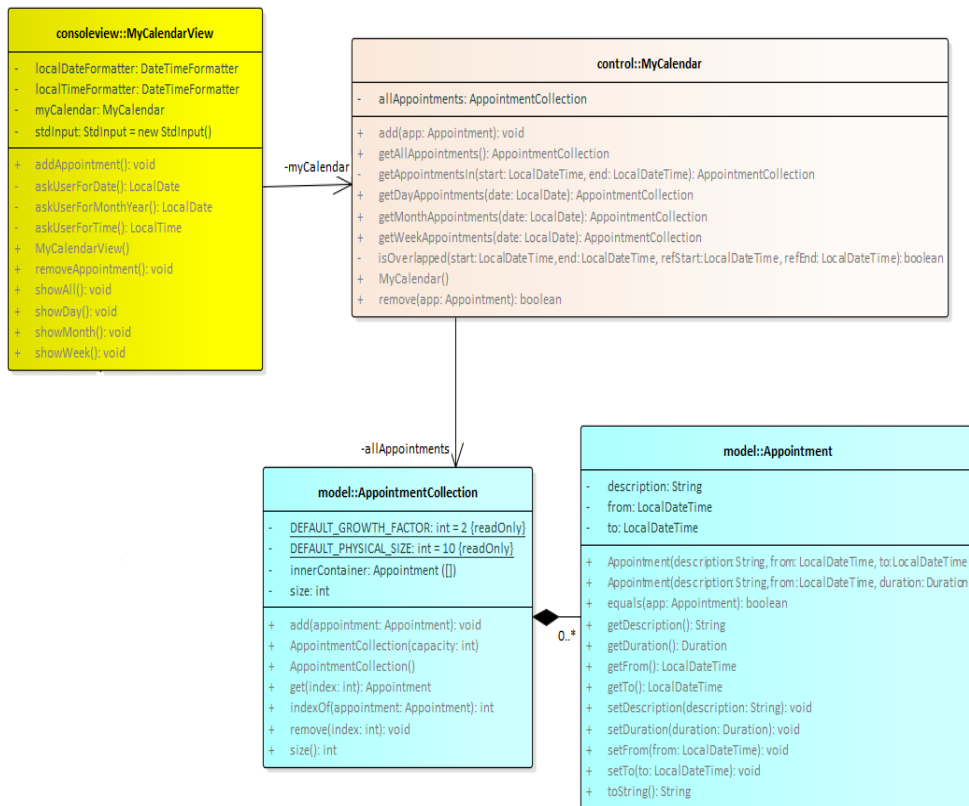


```
public class Triangolo {  
    private Vertice[] vertici;  
    public Triangolo(Vertice[] vertici) {  
        this.vertici = Arrays.copyOf(vertici, vertici.length);  
    }  
}
```

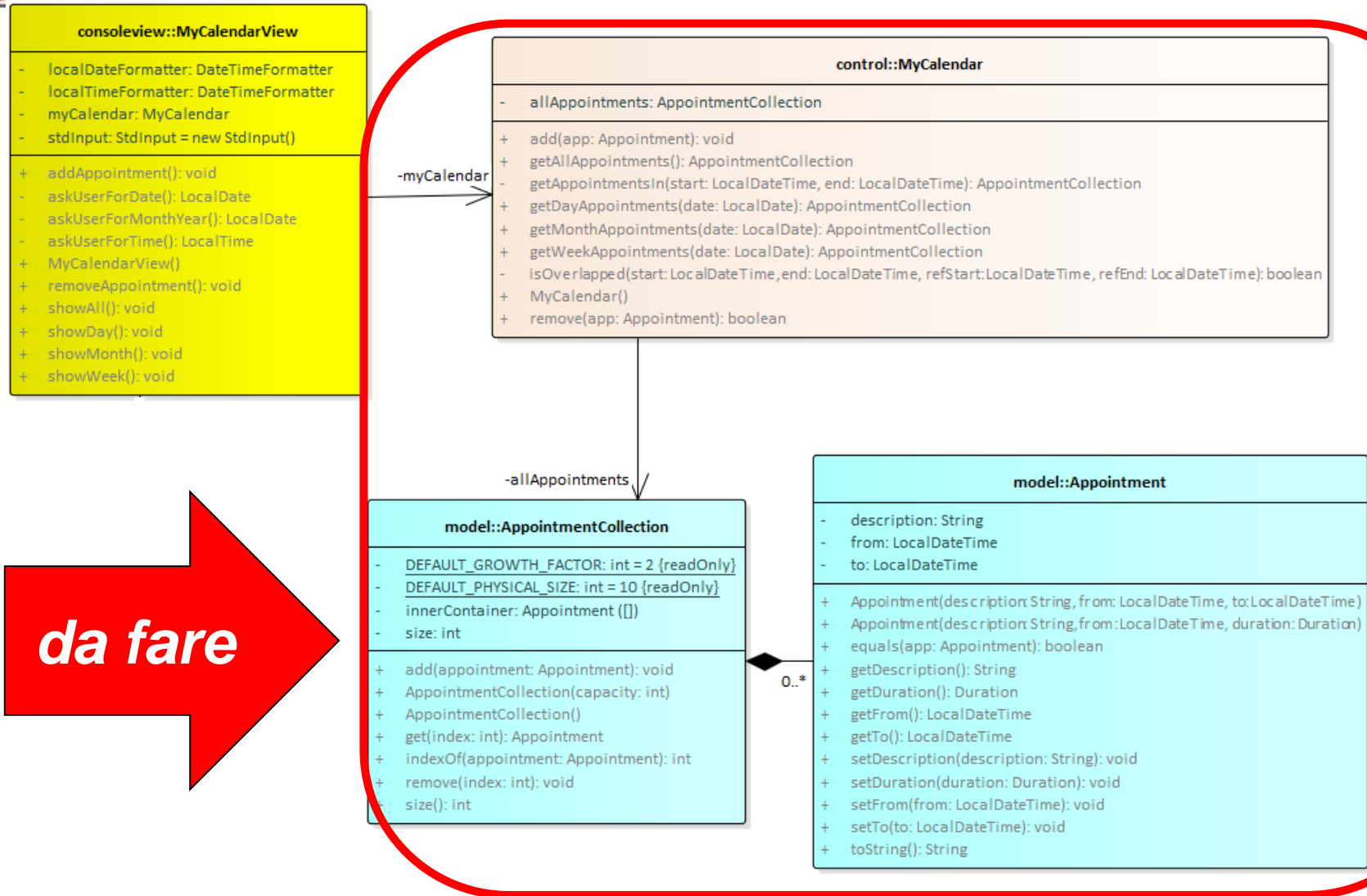
Occhio alla copia con strutture di grandi dimensioni...!!

UML - MyCalendar

- **MyCalendarView** ha un riferimento a **MyCalendar**
- **MyCalendar** ha un riferimento a **AppointmentCollection**
- **AppointmentCollection** ha un array di riferimenti ad **Appointment**



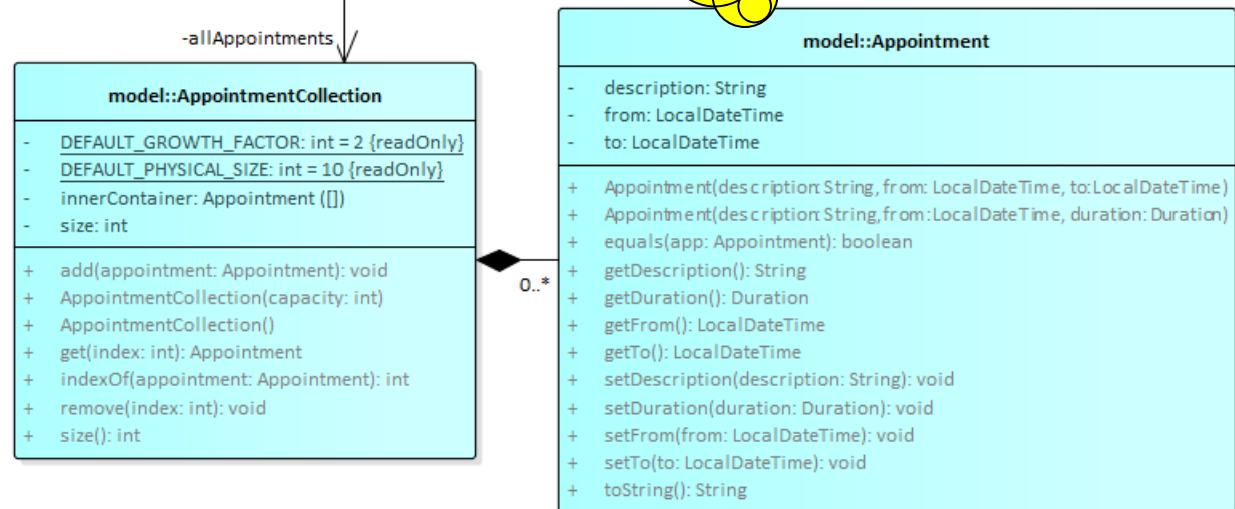
Organizzazione dell'applicazione



Organizzazione dell'applicazione

Suggerimento: in presenza di diverse classi da realizzare **si parte sempre da quella con meno legami** (dipendenze) con le altre classi, in questo caso si parte da **Appointment**

```
consoleview:  
- LocalDateForm  
- LocalTimeForm  
- ...  
- ask...  
- ...  
+ ...  
+ ...  
+ showDay(): void  
+ showMonth(): void  
+ showWeek(): void
```





Appointment (1/2)

- **Due costruttori** per costruire un appuntamento consistente
 - tre argomenti: descrizione, data/ora iniziale, data/ora finale
 - tre argomenti: descrizione, data/ora iniziale, durata
 - **NO costruttore di default** (un appuntamento "predefinito" non ha senso)
- **Accessor**
 - nel mondo reale è normale modificare appuntamenti già presi
 - stavolta abbiamo a che fare con un *oggetto modificabile*, non un valore
 - non solo recupero informazioni (**get***), ma anche modifica (**set***)
 - descrizione appuntamento: **getDescription**, **setDescription**
 - data/ora iniziale appuntamento: **getFrom**, **setFrom**
 - data/ora finale appuntamento: **getTo**, **setTo**
 - durata appuntamento: **getDuration**, **setDuration**
 - NB: la durata è *calcolata* a partire da data/ora iniziale e finale



Appointment (2/2)

- La *durata* è una proprietà calcolata
 - è la *differenza* fra data/ora iniziale e finale
 - precondizione: i relativi metodi possono funzionare solo se sono già impostate le date di inizio/fine appuntamento
 - i due metodi **getDuration** / **setDuration** devono fare calcoli
 - **getDuration** *calcola e restituisce* la durata dell'appuntamento
 - **setDuration** imposta la nuova durata dell'appuntamento mantenendo fissa la data/ora iniziale e *ricalcolando d'autorità la data/ora finale*
- Altri metodi: **toString** e **equals**
 - **toString** formatta le date usando un `DateTimeFormatter`
 - **equals** stabilisce che due appuntamenti sono uguali **SE... ? ☺**

NON INVENTARE SPECIFICHE! Seguire il progetto stabilito!

Appointment UML

model::Appointment	
-	description: String
-	from: LocalDateTime
-	to: LocalDateTime
+	Appointment(description: String, from: LocalDateTime, to: LocalDateTime)
+	Appointment(description: String, from: LocalDateTime, duration: Duration)
+	equals(app: Appointment): boolean
+	getDescription(): String
+	getDuration(): Duration
+	getFrom(): LocalDateTime
+	getTo(): LocalDateTime
+	setDescription(description: String): void
+	setDuration(duration: Duration): void
+	setFrom(from: LocalDateTime): void
+	setTo(to: LocalDateTime): void
+	toString(): String

NON INVENTARE SPECIFICHE! Seguire il progetto stabilito!



AppointmentCollection

- **AppointmentCollection** risolve lo stesso problema che **FractionCollection** risolveva con le frazioni → *identica*
- Si può riusare non solo il progetto, ma anche il codice
 - copiare, incollare e aggiustare sostituendo **Appointment** a **Frazione**
- Punti meritevoli di qualche attenzione:
 - *aggiungere/eliminare* un appuntamento
 - **add**: prende in ingresso un **Appointment** e lo aggiunge alla collezione
 - **remove**: prende in ingresso una posizione e rimuove l'appuntamento in quella posizione (attenzione alle rimozioni in «mezzo»)
 - *recuperare «quel certo» appuntamento*
 - **get**: riceve un indice e restituisce l'**Appointment** in quella posizione
 - **indexOf**: riceve un **Appointment** e ne restituisce la posizione (-1 se assente) [NB: per il confronto usare **equals** di **Appointment**]

AppointmentCollection

model::AppointmentCollection

- DEFAULT_GROWTH_FACTOR: int = 2 {readOnly}
 - DEFAULT_PHYSICAL_SIZE: int = 10 {readOnly}
 - innerContainer: Appointment ([])
 - size: int
-
- + add(appointment: Appointment): void
 - + AppointmentCollection(capacity: int)
 - + AppointmentCollection()
 - + get(index: int): Appointment
 - + indexOf(appointment: Appointment): int
 - + remove(index: int): void
 - + size(): int

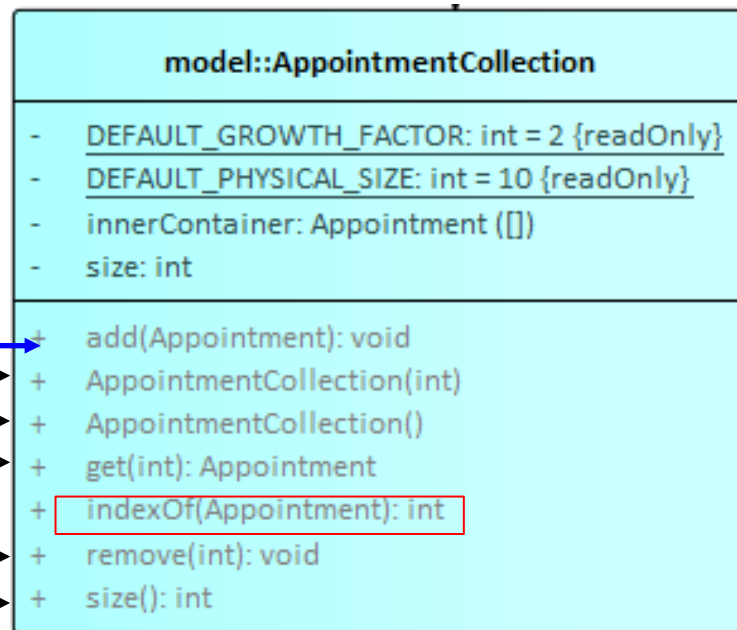
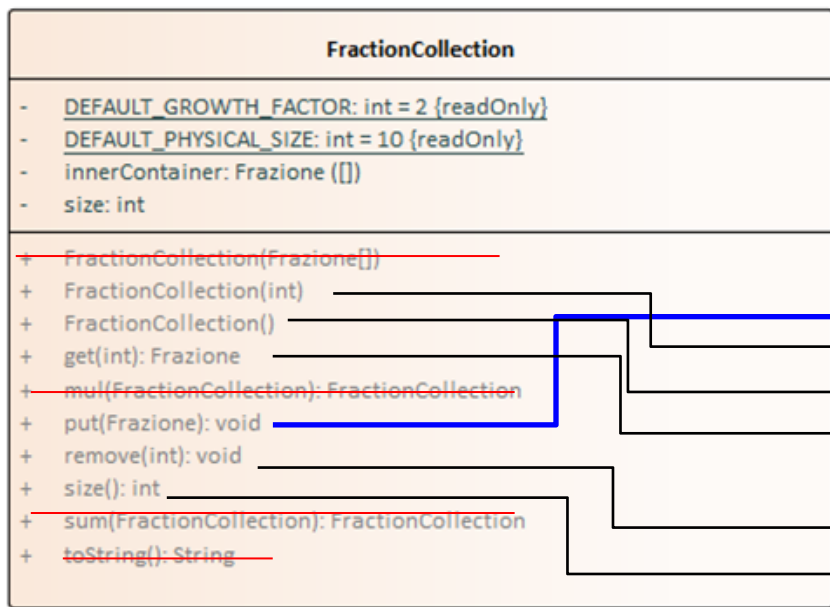
NON INVENTARE SPECIFICHE! Seguire il progetto stabilito!



AppointmentCollection

- **AppointmentCollection** → partite da **FractionCollection**
 1. Create la classe **AppointmentCollection** nello startkit che vi forniamo noi
 2. Copiate i vari metodi necessari da **FractionCollection** avendo cura di sostituire a **Frazione** la classe **Appointment** → attenzione abbiamo cambiato il nome al metodo **put** di **FractionCollection**, in **AppointmentCollection** si chiama **add**
 3. Aggiungete il metodo **indexOf(Appointment)** che non era presente in **FractionCollection**

AppointmentCollection



MyCalendar (1/4)

- **È il grande burattinaio della situazione**
 - è lui che manipola la **AppointmentCollection**
 - è lui che interagisce con la vista **MyCalendarView**
 - *tutto passa attraverso di lui* → vantaggi: sostituibilità, riusabilità, testabilità
- **Costruttore: solo default**
 - predispone le strutture dati → crea la **AppointmentCollection**
- **Metodi**
 - *aggiungere/eliminare un appuntamento*
 - **add**: riceve un appuntamento e tenta di aggiungerlo alla collezione;
 - **remove**: riceve un appuntamento e tenta di eliminarlo dalla collezione; restituisce un **boolean** che indica se l'eliminazione ha avuto successo
 - *ottenere l'elenco di tutti gli appuntamenti*
 - **getAllAppointments**: restituisce una copia della collezione di appuntamenti

MyCalendar (2/4)

- **Metodi (segue)**

- *ottenere l'elenco degli appuntamenti giornalieri*
 - **getDayAppointments**: prende in ingresso un oggetto **LocalDate** che rappresenta un certo **giorno** e restituisce *tutti gli appuntamenti di quel giorno*
→ attenzione agli appuntamenti che durano più giorni...!
- *ottenere l'elenco degli appuntamenti mensili*
 - **getMonthAppointments**: idem: prende in ingresso un **LocalDate** che rappresenta un certo **me**se e restituisce *tutti gli appuntamenti di quel mese*
→ attenzione agli appuntamenti che durano più mesi...!

Se un appuntamento dura da lunedì a mercoledì, va incluso anche nell'elenco di appuntamenti del martedì!

Se un appuntamento dura da marzo a settembre, va incluso anche nell'elenco di appuntamenti di aprile, maggio, giugno...!

Quali algoritmi?



MyCalendar (3/4)

ATTENZIONE:

- i metodi **getDayAppointments** e **getMonthAppointments** prendono in ingresso una **LocalDate**

MA

- date/orari iniziale e finale degli appuntamenti sono **LocalDateTime**

QUINDI

- per verificare se un appuntamento è nel giorno o mese che interessa occorrerebbe poter confrontare (**isEqual**, **isBefore**, **isAfter**) dei **LocalDateTime** con dei **LocalDate** ...
- **.. peccato che non si possa fare!**
 - sono classi diverse, con granularità temporali diverse
 - *come potremmo confrontare un "giorno" con un "giorno & orario" ?*

E ALLORA...?

MyCalendar (4/4)

- Poiché i confronti si possono fare solo fra oggetti omogenei, occorre in qualche modo "convertire" un **LocalDate** in una **coppia di LocalDateTime**
- Un possibile modo di farlo è *definire un apposito intervallo*:
 - per un giorno:

[Inizio del Giorno, Inizio del Giorno successivo)

Chiuso a sinistra...

...aperto a destra

- per un mese:

[Inizio del Mese, Inizio del Mese successivo)

Chiuso a sinistra...

...aperto a destra

- Perché intervalli chiusi a sinistra e aperti a destra?
- Perché così è più facile determinare «la fine» dell'intervallo: altrimenti dovremmo determinare *l'istante prima* della fine della giornata..!

MyCalendar UML

control::MyCalendar

```
- allAppointments: AppointmentCollection  
  
+ add(app: Appointment): void  
+ getAllAppointments(): AppointmentCollection  
- getAppointmentsIn(start: LocalDateTime, end: LocalDateTime): AppointmentCollection  
+ getDayAppointments(date: LocalDate): AppointmentCollection  
+ getMonthAppointments(date: LocalDate): AppointmentCollection  
+ getWeekAppointments(date: LocalDate): AppointmentCollection  
- isOverlapped(start: LocalDateTime, end: LocalDateTime, refStart: LocalDateTime, refEnd: LocalDateTime): boolean  
+ MyCalendar()  
+ remove(app: Appointment): boolean
```

NON INVENTARE SPECIFICHE! Seguire il progetto stabilito!

Algoritmica (1/5)

COME filtrare gli appuntamenti?

- Per estrarre appuntamenti relativi a un qualsiasi intervallo temporale (giorno, mese..), occorre:

1. Determinare l'intervallo di tempo che interessa, nella forma stabilita

`[LocalDateTime inizio, LocalDateTime fine)`

Chiuso a sinistra...

...aperto a destra

→ algoritmo identico per giorni, mesi, anni, settimane...

2. Estrarre gli appuntamenti "rilevanti" per quell'intervallo di tempo

→ l'appuntamento ci interessa (e quindi va restituito in uscita)

SOLO SE c'è una *sovrapposizione* fra

- l'intervallo dell'appuntamento (es. dura da lunedì a mercoledì)
- l'intervallo di estrazione (estrarre gli appuntamenti fra martedì e giovedì)



Algoritmica (2/5)

- **Determinare l'intervallo: GIORNO**

- Da specifica, arriva come parametro un oggetto **LocalDate** [**getDayAppointments(LocalDate)**] che rappresenta il giorno che interessa
- da lì occorre calcolare gli istanti di **inizio** e **fine** giornata
 - l'istante di inizio della giornata
 - creare un **LocalDateTime** con ore, minuti, secondi (...) a zero.
 - l'istante di fine della giornata
 - creare un **LocalDateTime** aggiungendo 1 giorno al precedente

E se in mezzo c'è il **cambio ora solare/legale?**

Nessun problema: «lui» sa cosa si deve fare!

Algoritmica (3/5)

- **Determinare l'intervallo: MESE**

- Da specifica, arriva come parametro un oggetto **LocalDate** [**getMonthAppointments(LocalDate)**] che rappresenta **un giorno a caso all'interno del mese che interessa**
- da lì occorre calcolare gli istanti di **inizio** e **fine** giornata
 - l'istante di inizio del mese
 - creare un **LocalDateTime** con il giorno a 1 e ore, minuti, secondi (...) a zero.
 - l'istante di fine del mese
 - creare un **LocalDateTime** aggiungendo 1 mese al precedente

- **Determinare l'intervallo: SETTIMANA**

- analogo al mese: da specifica, arriva come parametro un oggetto **LocalDate** [**getWeekAppointments(LocalDate)**] che rappresenta **un giorno a caso all'interno della settimana che interessa**

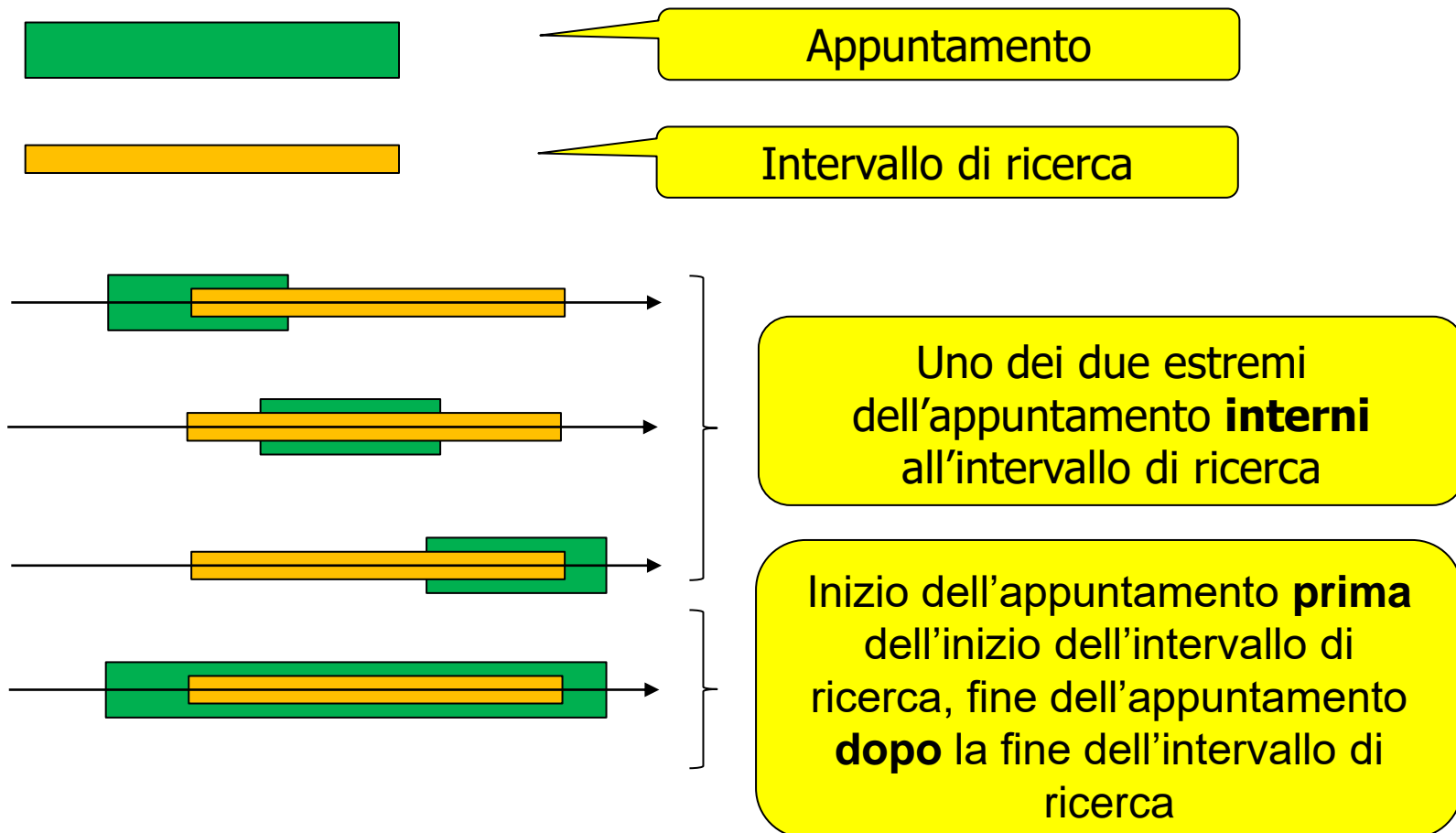


Algoritmica (4/5)

- **Estrarre gli appuntamenti "rilevanti" per quell'intervallo**
 - algoritmo identico in tutti i casi
 - conta solo il concetto di *sovrapposizione* fra
 - l'intervallo dell'appuntamento (es. dura da lunedì a mercoledì)
 - l'intervallo di estrazione (estrarre gli appuntamenti fra martedì e giovedì)
- È un problema da Fondamenti T-1!
 - dovrete già saperlo fare: è questione di logica e ragionamento
 - un disegno è fondamentale per
 - *analizzare bene il problema*
 - *catturare tutti i casi*
 - predisporre quindi il corrispondente *collaudo*
 - **MA se proprio serve un aiutino...**

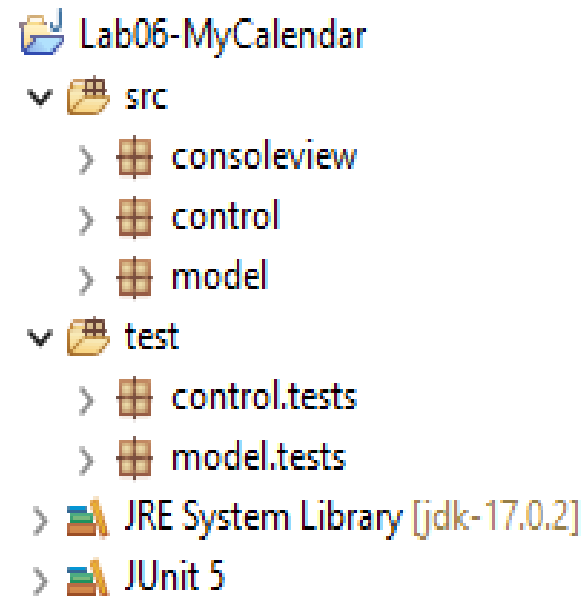
Algoritmica (5/5)

- Sovrapposizione di intervalli: quattro casi possibili**



Package & Start kit

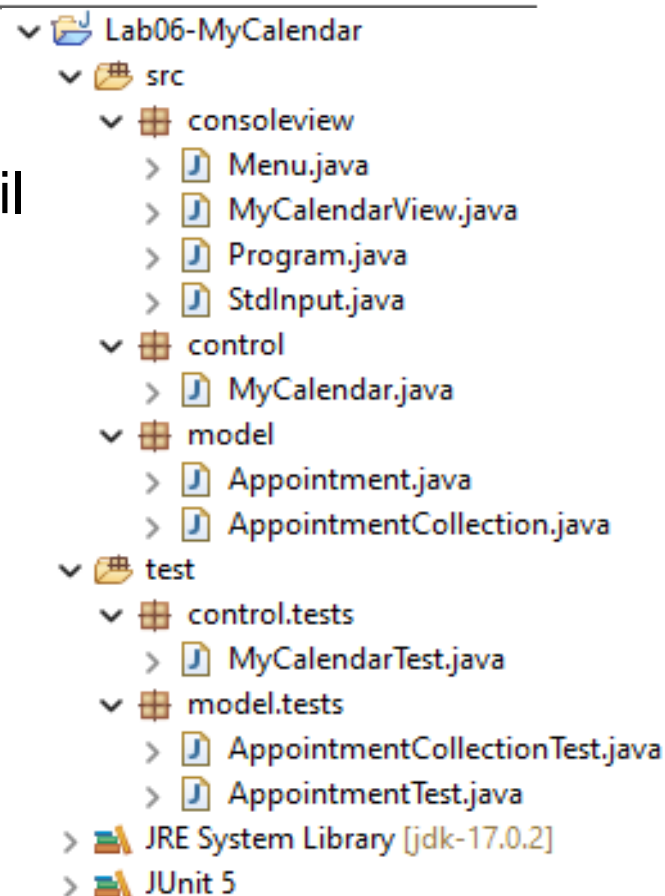
- Strutturiamo il nostro progetto seguendo la separazione logica suggerita dal Pattern MVC
- **Due diversi source folder** per separare i sorgenti e le classi di test
→ una buona organizzazione dell'ambiente di lavoro aiuta anche chi verrà dopo di noi ad «orientarsi» meglio nel nostro lavoro
- Il folder «test» lo trovate già pronto nello startkit con i test JUnit per testare le vostre classi



OCCHIO ai nomi dei package, delle classi e dei metodi, altrimenti i test ☹

Package & Start kit

- I nostri package sono:
 - **control**: contiene **MyCalendar**, il grande burattinaio
 - **model**: contiene **Appointment** e **AppointmentCollection**, il modello della nostra applicazione
 - ...



OCCHIO ai i nomi dei package, delle classi e dei metodi, altrimenti i test ☹

Package & Start kit

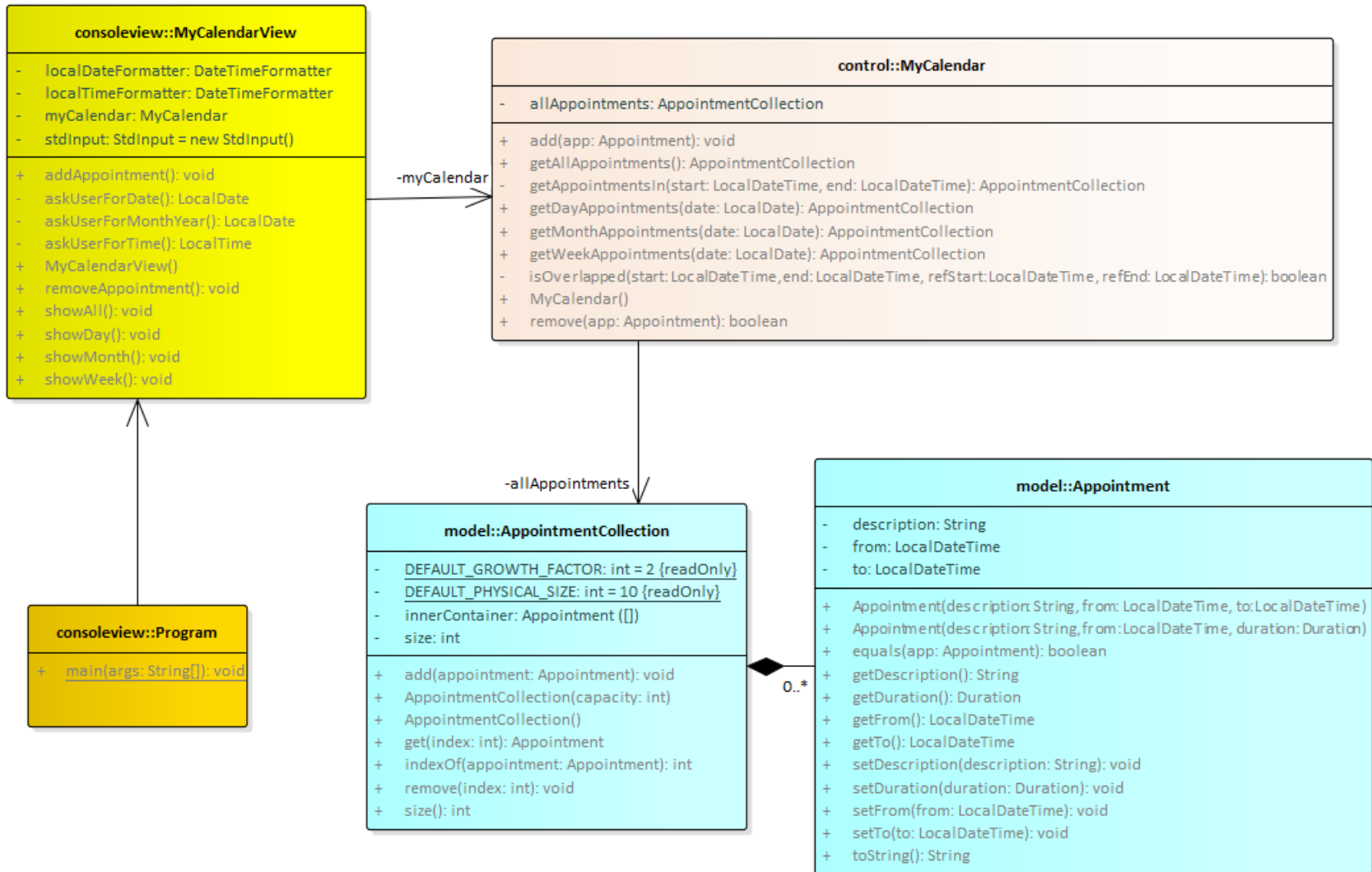
– consoleview:

- contiene la classe **MyCalendarView** fornita già pronta, nonché una classe ausiliaria **Menu** usata per realizzare un menù di scelta su console
- una classe **StdInput** permette di recuperare i dati immessi dall'utente console
- la classe **Program** con il main per l'esecuzione

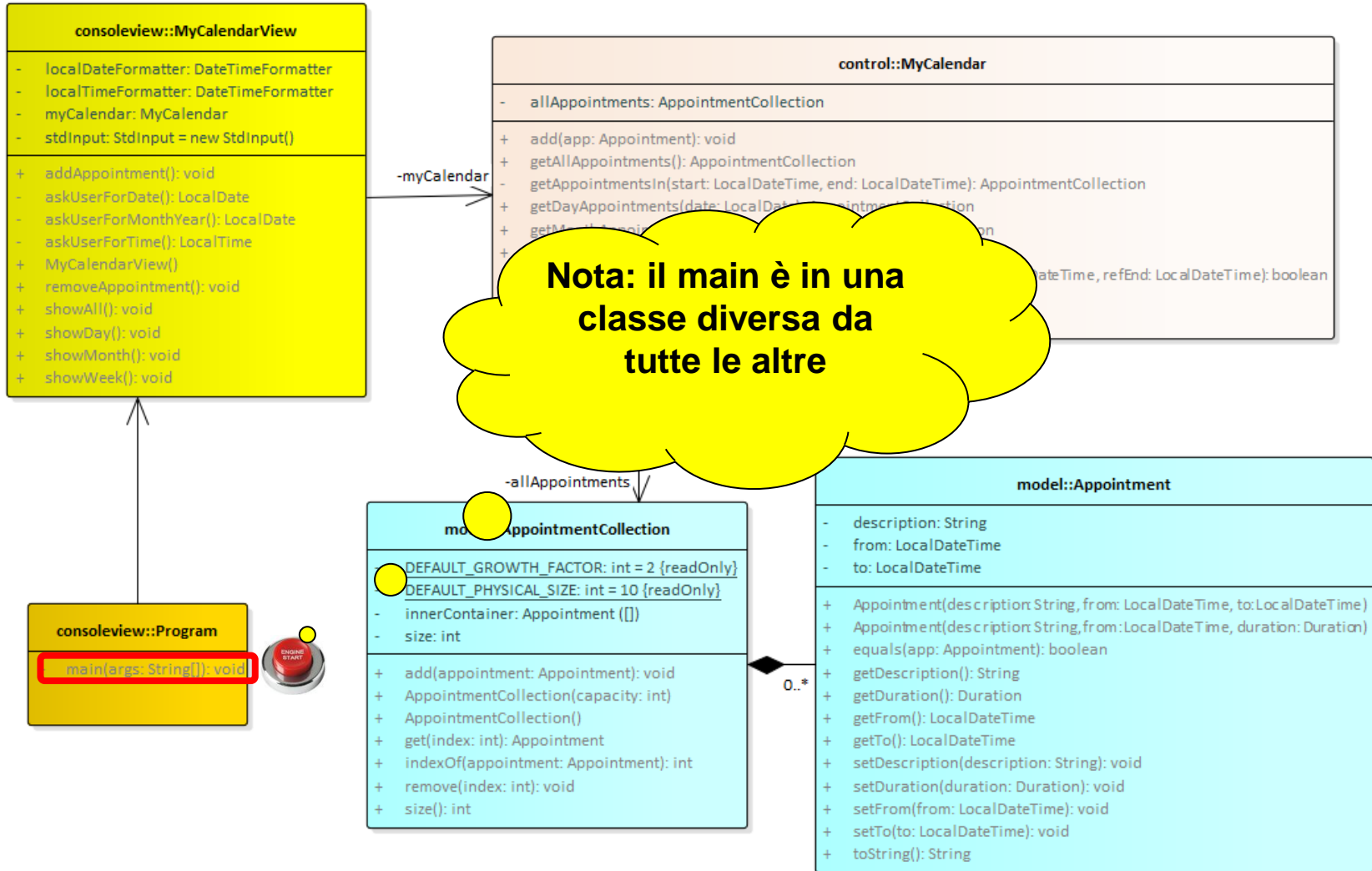


OCCHIO ai nomi dei package, delle classi e dei metodi, altrimenti i test ☹

Organizzazione dell'applicazione



Organizzazione dell'applicazione





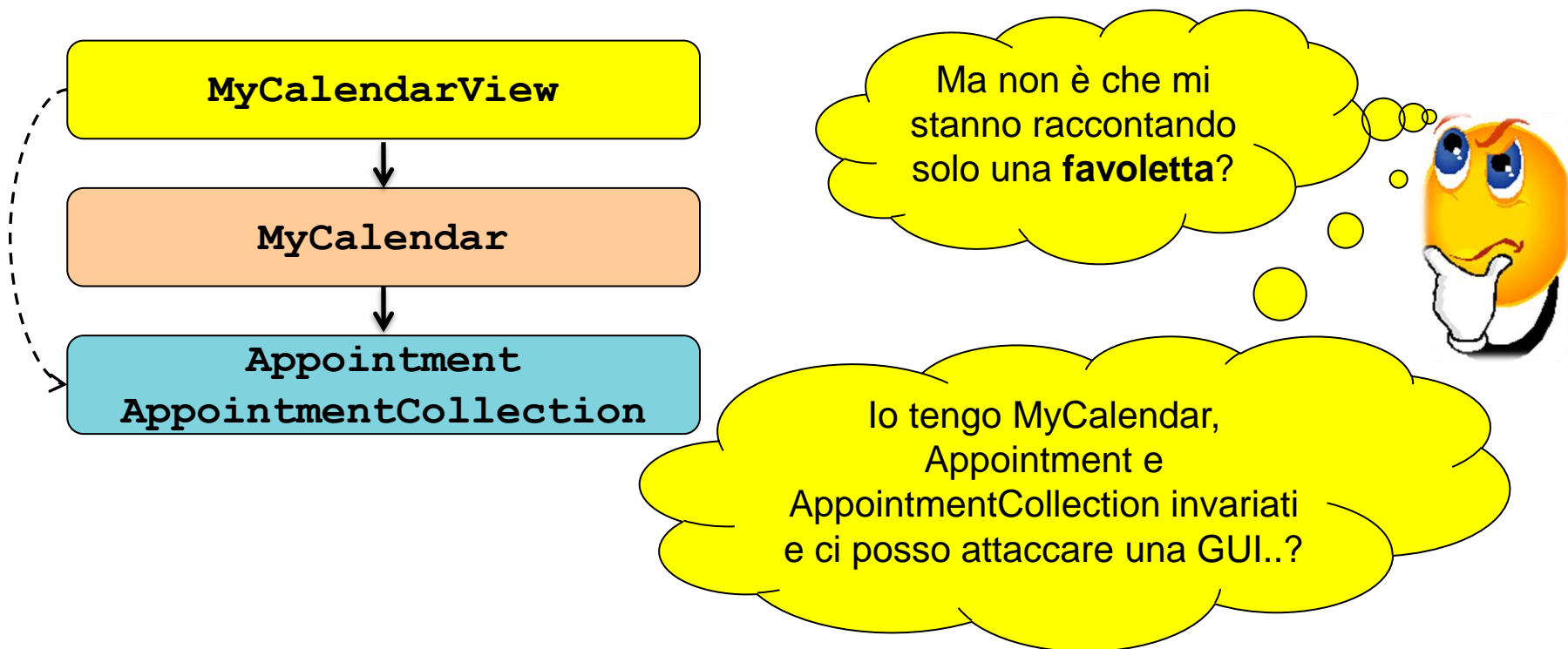
Cosa dovete fare

- Realizzare e collaudare nell'ordine le tre classi:
 - **Appointment**
 - **AppointmentCollection**
 - **MyCalendar**
- Eventuale funzionalità opzionale:
 - ottenere la lista degli appuntamenti settimanali

Buon Lavoro!!!!

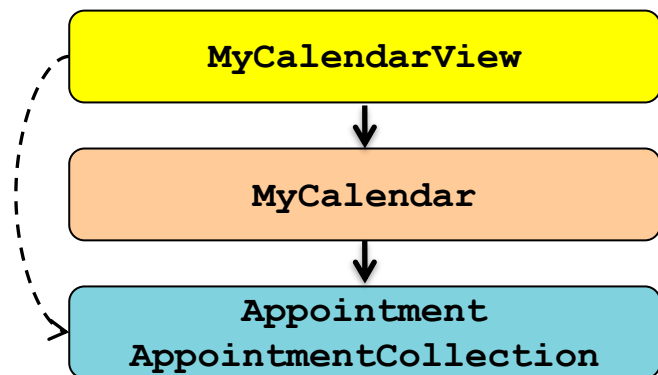
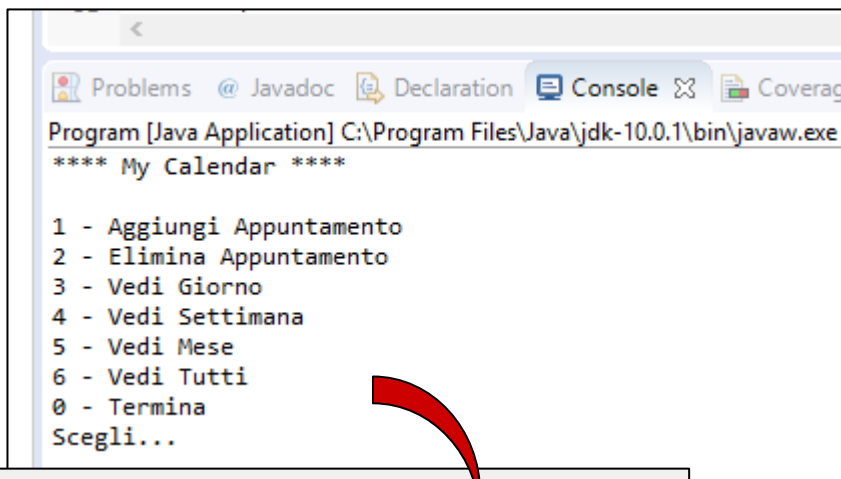
Model-View-Controller: funziona davvero...?

Calendario appuntamenti con MVC



Ebbene sì! Se tutto è progettato bene, dev'essere possibile cambiare la *View senza colpo ferire* e ottenere una visualizzazione diversa 😊

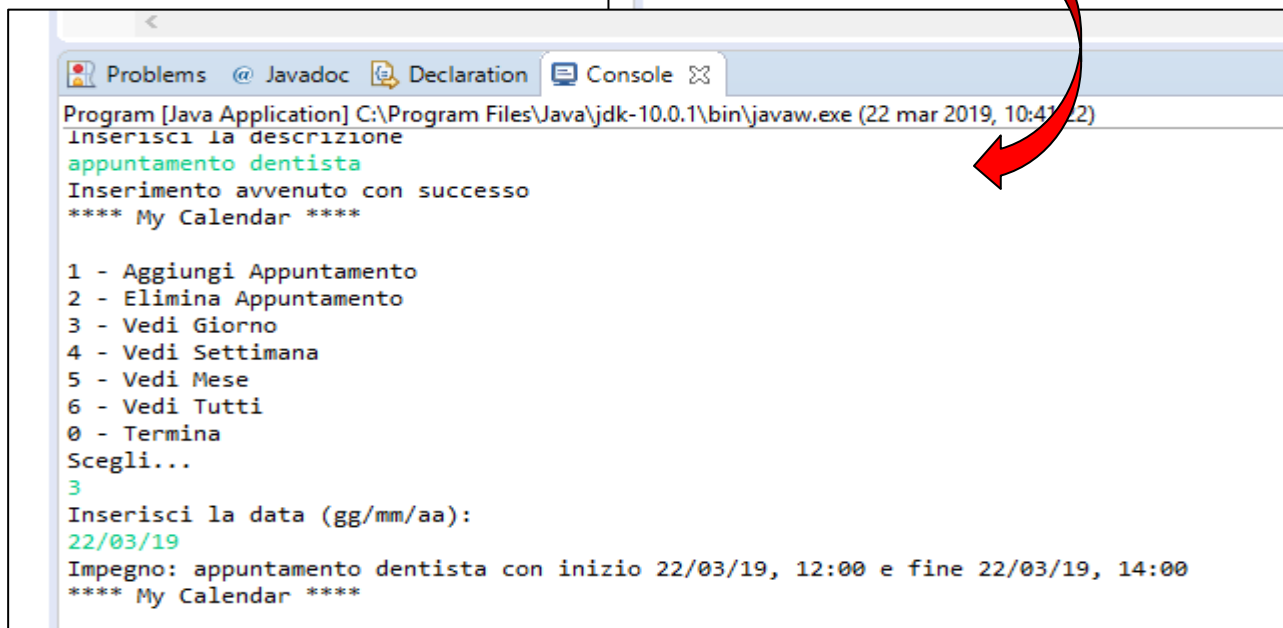
Da I/O su Console...

```

Program [Java Application] C:\Program Files\Java\jdk-10.0.1\bin\javaw.exe
**** My Calendar ****

1 - Aggiungi Appuntamento
2 - Elimina Appuntamento
3 - Vedi Giorno
4 - Vedi Settimana
5 - Vedi Mese
6 - Vedi Tutti
0 - Termina
Scegli...
  
```



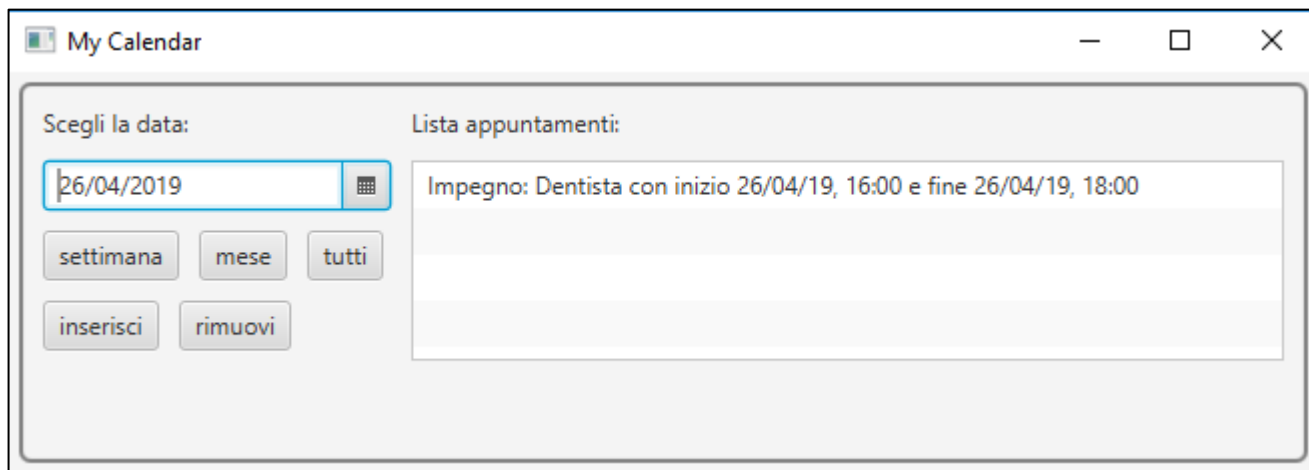
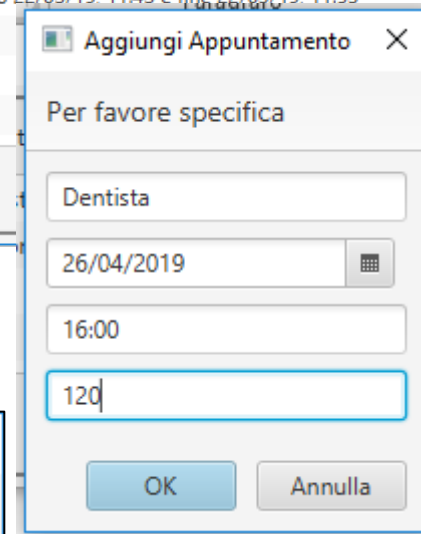
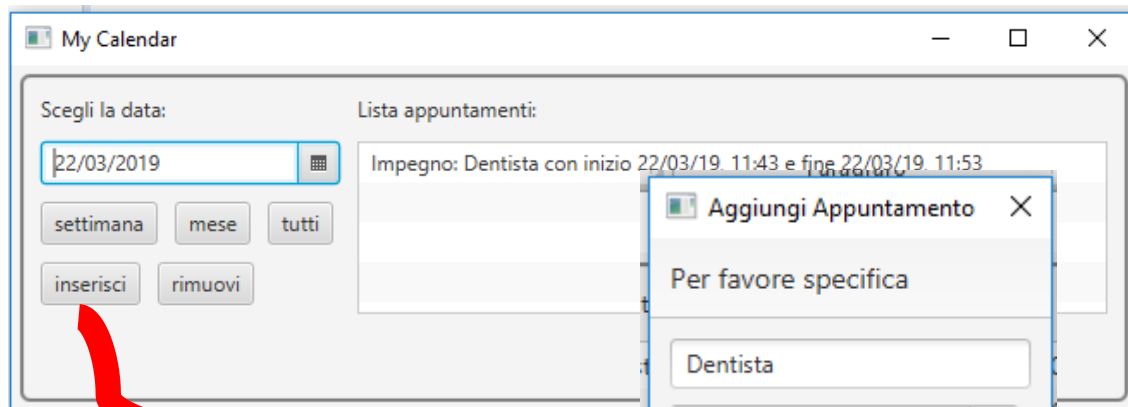
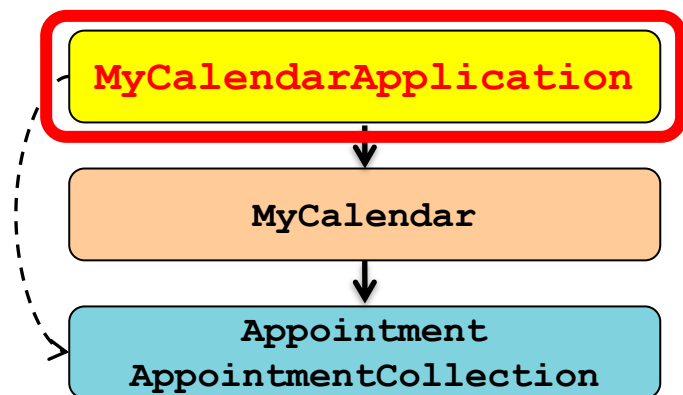
```

Program [Java Application] C:\Program Files\Java\jdk-10.0.1\bin\javaw.exe (22 mar 2019, 10:41:22)
Inserisci la descrizione
appuntamento dentista
Inserimento avvenuto con successo
**** My Calendar ****

1 - Aggiungi Appuntamento
2 - Elimina Appuntamento
3 - Vedi Giorno
4 - Vedi Settimana
5 - Vedi Mese
6 - Vedi Tutti
0 - Termina
Scegli...
3
Inserisci la data (gg/mm/aa):
22/03/19
Impegno: appuntamento dentista con inizio 22/03/19, 12:00 e fine 22/03/19, 14:00
**** My Calendar ****
  
```

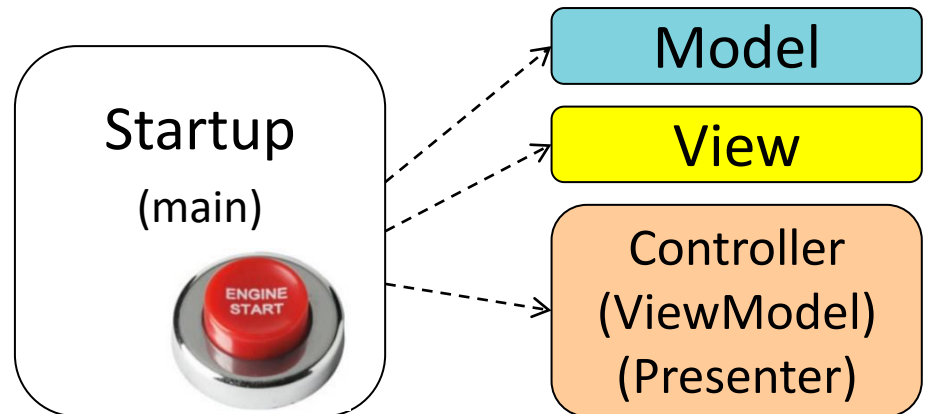
A red curved arrow points from the 'Scegli...' prompt in the top screenshot to the '3' input in the bottom screenshot, indicating the user's selection of the 'Vedi Giorno' option.

... a una vera GUI !



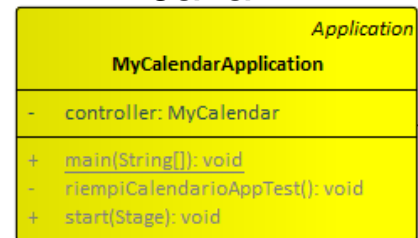
Organizzazione dell'applicazione

- Si era detto: **non preoccupiamoci di "dove sia" il main**
 - non è parte dell'organizzazione MVC e c'è un ottimo motivo!
 - dato che il suo scopo è *gestire tutta la fase di startup*, **può cambiare** (anzi: ci aspettiamo che cambi!) **se qualche componente è diverso**
 - nel nostro caso, *è cambiata totalmente la view* → è molto probabile che la classe che gestisce lo startup debba *fare cose piuttosto diverse da prima*
- Ma in ogni caso, resta fermo il principio: essa deve
 - creare la struttura
 - connettere le parti
 - lanciare l'applicazione
- *indipendentemente dal modo concreto di farlo*

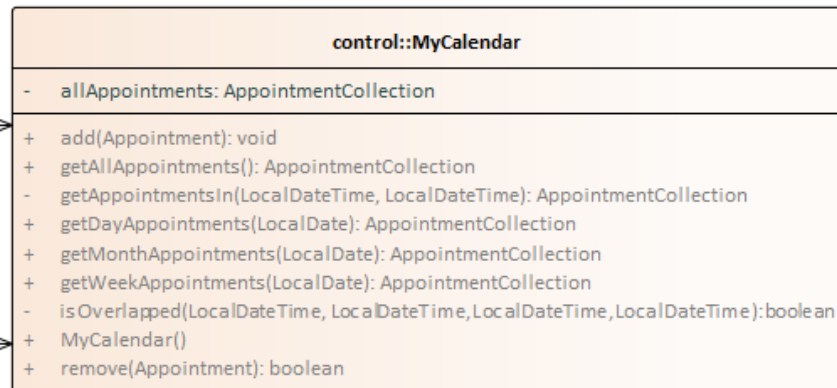
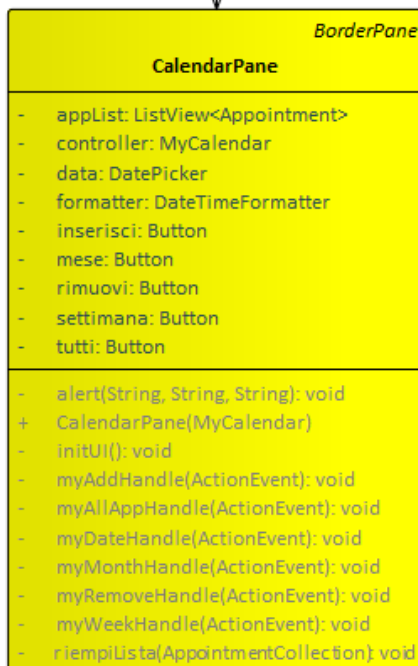


Una vera app grafica: struttura

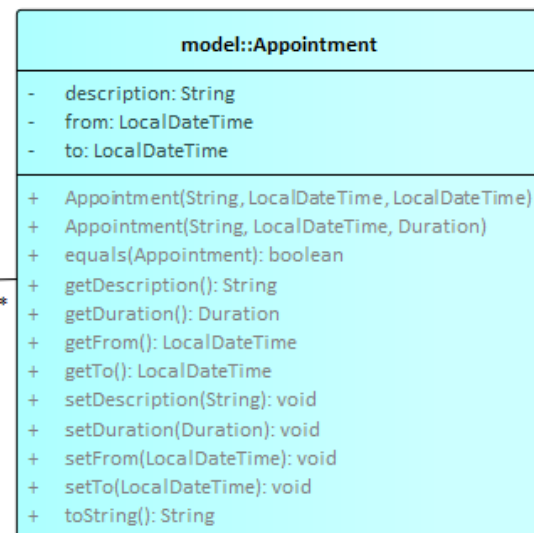
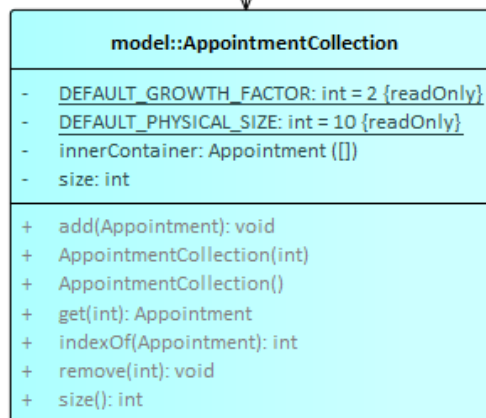
JavaFx



«use»



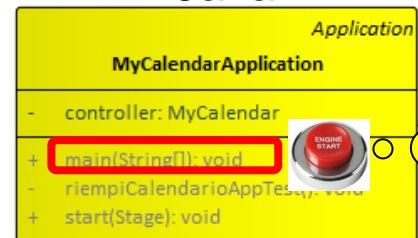
-allAppointments



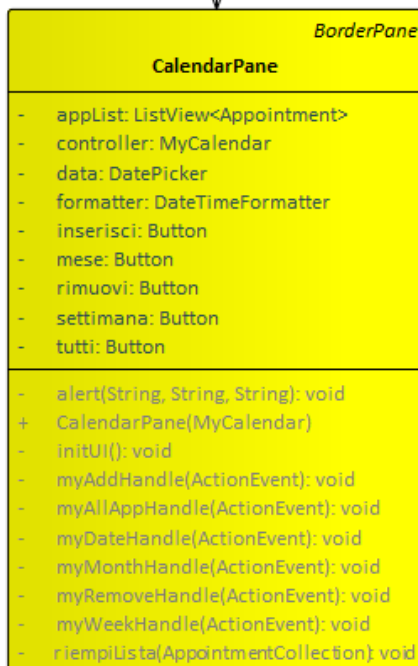
Le classi
di prima

Una vera app grafica: struttura

JavaFx



«use»



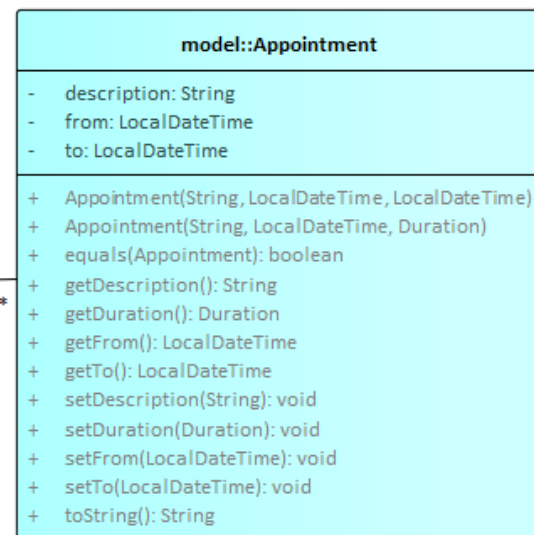
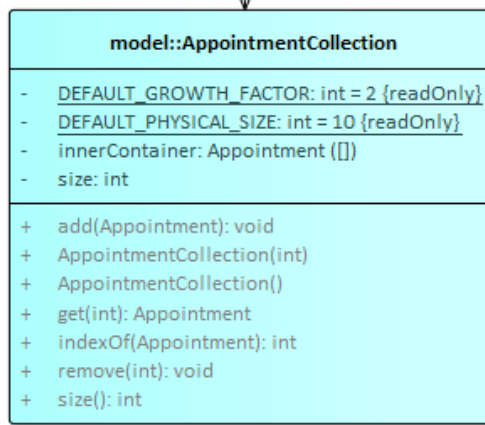
Nota: il main ora è nella UI !

Evidentemente, è un luogo
"appropriato" nell'architettura
JavaFX... ☺

classi
di prima



-allAppointments



0..*