# Fondamenti di Informatica T2
# Lab07 – il caso di studio Phone Plan

*Corso di Laurea in Ingegneria Informatica*

Anno accademico 2021/2022

**Prof. ROBERTA CALEGARI**

**Prof. AMBRA MOLESINI**

*Dipartimento di Informatica – Scienza e Ingegneria (DISI)*

# Tariffazione Telefonica

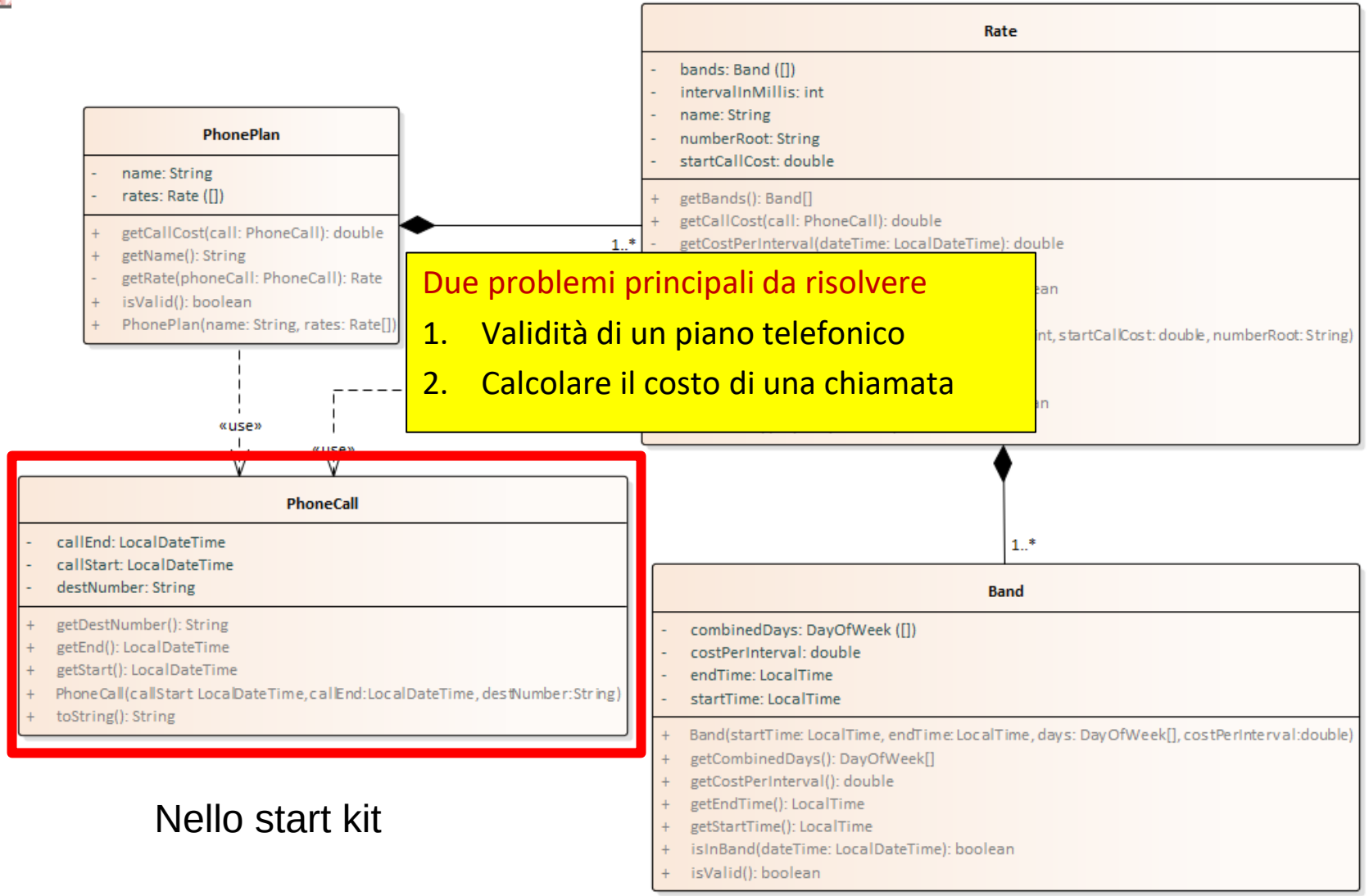
OBIETTIVO

modellare un sistema di tariffazione telefonica

- Progettare il **piano telefonico**
- Dal piano telefonico si calcola il costo di una **chiamata** sulla base di:
  - istanti di inizio/fine chiamata
  - numero chiamato
    (il costo può dipendere dall'operatore, dal paese di destinazione, etc.)
- Il numero chiamato determina la **tariffa** applicata
- La tariffa è basata su determinate **fasce orarie e giorni**

# Parte II: algoritmica

# Modello del dominio




Nello start kit

# Validità piano telefonico

- Il **Piano telefonico** è valido se ogni sua Tariffa (`Rate`) è valida
- La **Tariffa** è valida se ha copertura su ogni giorno/ora e se ogni Fascia è valida
  → *ad esempio, quella sotto non lo è!!*
- La **Fascia** (`Band`) è valida se sia l' orario di inizio e fine sia i giorni sono «plausibili»

```
Tariffa UU scatto alla risposta 12 cent intervalli di 1000 ms
applicabile a tutti i numeri che iniziano con "+39339"
    SAB-DOM 10 cent ad intervallo sempre
    LUN-MER 08:00-18:30 20 cent ad intervallo
    LUN-VEN 18:30-08:00 10 cent ad intervallo
```

**MER-VEN 08:00-18:30 non risulta coperto!**

# Esempio calcolo costo (1)

- Calcolare il costo di una **Chiamata** effettuata interrogando il **Piano telefonico**

```
MERCOLEDÌ dalle 22:24 alle 22:27 al numero "+393339312312"
```

```
PhoneCall call = new PhoneCall(start, end, "+393339312312");
double cost = plan.getCallCost(call);
```

- Ricerco la **Tariffa** da applicare in base alle cifre iniziali del numero e *delego* alla Tariffa il calcolo del costo della chiamata
- Tariffa a sua volta interrogherà la **Fascia** corretta per il costo ad intervallo

```
Tariffa UU scatto alla risposta 12 cent intervalli di 1000 ms
applicabile a tutti i numeri che iniziano con "+39339"
    SAB-DOM 10 cent ad intervallo sempre
    LUN-VEN 08:00-18:30 20 cent ad intervallo
    LUN-VEN 18:30-08:00 10 cent ad intervallo
```

NB: qui i "soldi" sono € **cent**

```
Durata 3 minuti = 180000 ms → 180000/1000 = 180 intervalli
Costo chiamata = 12 + 180*10 = 1812
```

Costo intervallo nella seconda fascia oraria

# Esempio calcolo costo (2)

- Calcolare il costo di una **Chiamata** effettuata interrogando il **Piano telefonico**

MERCOLEDÌ dalle 18:28:10 alle 18:32:25 al numero "+393339123123 12"

NB: chiamata a cavallo di due fasce orarie

Tariffa **UU** scatto alla risposta **15 cent** intervalli di **1 minuto = 60000 ms** applicabile a tutti i numeri che iniziano con "+39339"

SAB-DOM 10 cent ad intervallo sempre

LUN-VEN 08:00-18:30 20 cent ad intervallo

LUN-VEN 18:30-08:00 10 cent ad intervallo

Durata 4 minuti e 15 secondi = 255000 ms → 4,25 intervalli → 5 intervalli

Costo chiamata = 15 + 5*20 = **115**

Semplificazione: si usa *per tutta la chiamata* il costo della *fascia iniziale* della chiamata

Attenzione all'arrotondamento

# Esprimere il costo

- L'analisi precedente ha parlato sempre di *costo*
  *senza specificare come misurarlo*
- Ovviamente, un costo è espresso in una qualche *unità di misura:*
  vogliamo / dobbiamo specificarla ora, *o è irrilevante?*
  - se volessimo generare stringhe (o fatture..) legate al Paese e alla cultura locale, dovremmo saperlo…
  - .. ma per fare i calcoli, non è necessario: *la logica è invariante!*
- Stabiliamo perciò di prescindere dalla specifica valuta
  → prezzi e costi siano espressi in *"soldi"*
  - non ha importanza cosa siano realmente, ai fini del calcolo

# Gli algoritmi di Rate e PhonePlan

- Grazie alla forte strutturazione, gli algoritmi sono a loro volta ben incapsulati → facilmente testabili
  1. Calcolo del costo di una chiamata
     a) `PhonePlan.getCallCost`
     b) `Rate.getCallCost`
  2. Verifica di validità del piano telefonico
     a) `PhonePlan.isValid`
     b) `Rate.isValid`
- La struttura dei dati (guarda caso…) *agevola e promuove* l'architettura degli algoritmi

Regola aurea:
*più struttura = meno codice negli algoritmi*

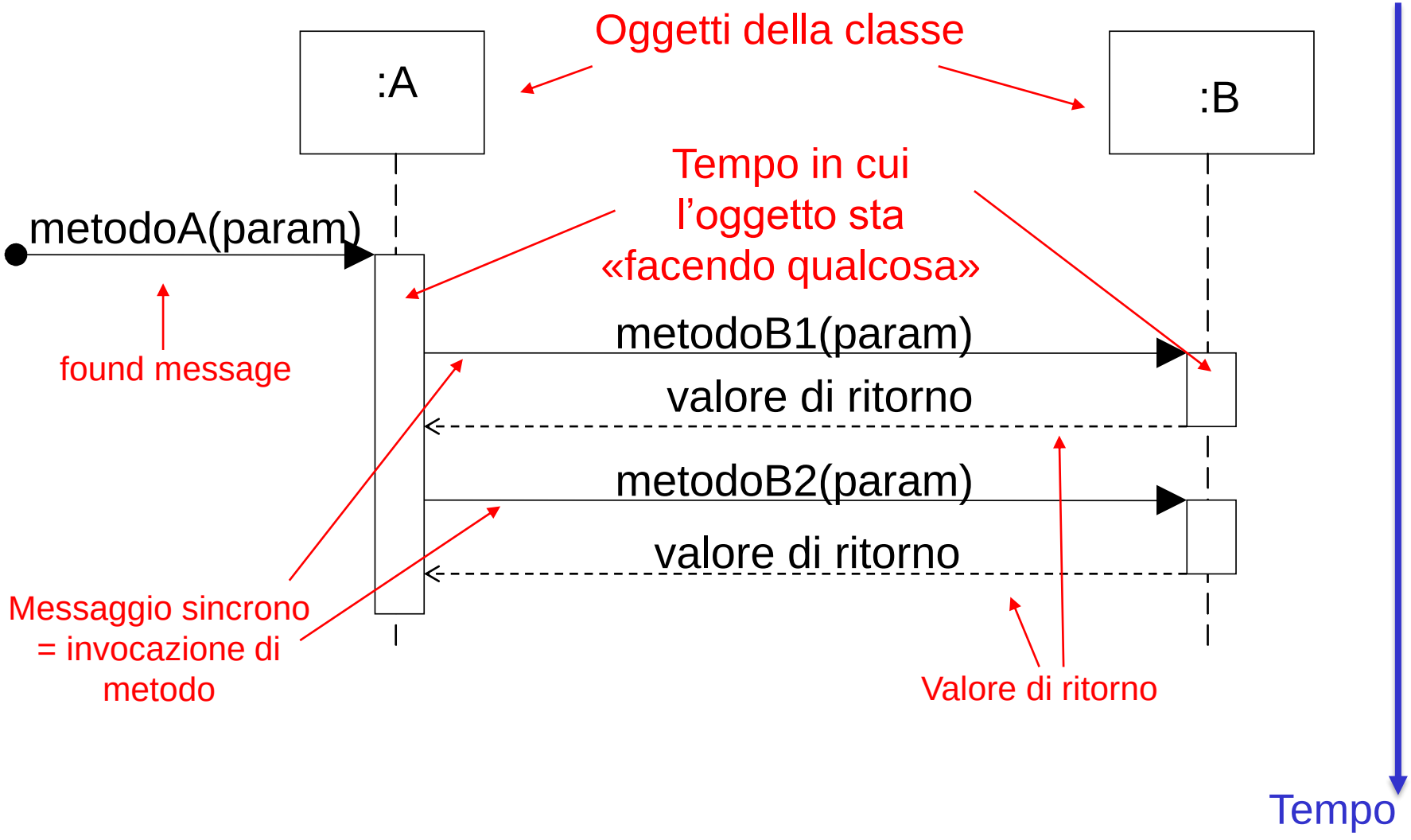
# How we did it

- I metodi *privati* che compaiono nell'UML fanno parte di *una possibile* soluzione (la nostra..), che *non è l'unica possibile*
- Approccio TOP-DOWN
  - suddividere il problema in sotto-problemi
  - risolverli uno alla volta, singolarmente , incapsulandoli ciascuno in un opportuno metodo (di norma privato)
  - se necessario, ogni sotto-problema può essere ri-scomposto, e così via
- Avere un metodo privato che viene invocato in un solo punto del codice è *normale* e ***non costituisce un problema!***
  - non andiamo al "risparmio" su metodi e nomi!
  - i metodi servono anche per «dare un nome» ad una porzione di codice di cui altrimenti sarebbe difficile intuire lo scopo
  - cercate di volere bene a chi, dopo di voi, metterà le mani sul vostro codice… e lui vorrà bene a voi.

# Algoritmi: aspetto dinamico

- Per seguire il funzionamento di questi algoritmi la sola vista strutturale non è sufficiente: *LA STRUTTURA NON È TUTTO!*
- Ci serve un'altra «vista» che illustri la *DINAMICA*
  - «interazione» = scambio di messaggi ( « chi chiama chi» )
- A tale scopo si sfrutta un *altro tipo di diagramma UML*
  - «interazione» = **DIAGRAMMA DI SEQUENZA**
- Fotografa il sistema a *run time*
  (non la struttura come il diagramma delle classi)
  → specifica come interagiscono tra loro le parti del sistema
- **NB: conoscerlo NON è obiettivo di questo corso**
  - semplicemente, lo usiamo in alcuni laboratori perché fa comodo..
  - .. e in questo caso è alquanto "intuitivo"

# Diagramma di Sequenza



# Diagramma di Sequenza

- La specifica di UML permette di esprimere comportamenti più complessi rispetto al singolo scambio di messaggi
- A tale scopo sono messi a disposizione i Combined Fragment
  - contenitori atti a delimitare un'area d'interesse nel diagramma
  - servono a spiegare che una certa catena di eventi, racchiusa in uno o più operandi, si verificherà in base alla semantica dell'operatore associato
- Ogni fragment ha un operatore ed una eventuale *guardia*
- Nei nostri diagrammi ne vedremo due:
  - Loop: rappresenta un *ciclo* (for/while/repeat)
  - Alt: rappresenta *alternative* (if-then-else)

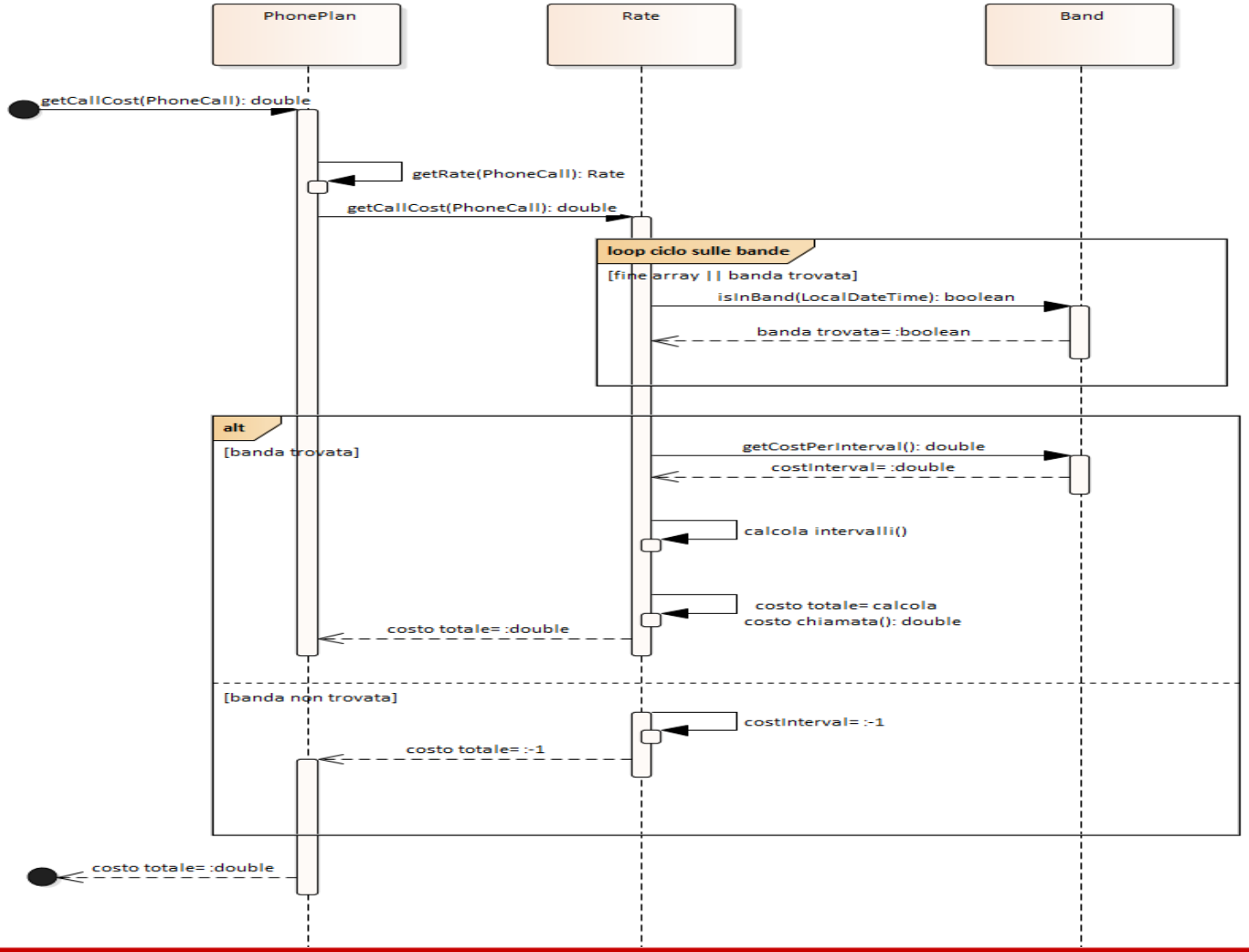
# Algoritmo 1 (a)
# Calcolo costo chiamata

- **`PhonePlan.getCallCost`**
  - **Input**: la chiamata, ovvero:
    - istanti di inizio/fine chiamata
    - numero chiamato
  - **Come agire:**
    1. Selezionare una **tariffa applicabile**
    2. Chiedere a quella tariffa di calcolare il **costo** della chiamata → DELEGA

# Algoritmo 1 (b)
# Calcolo costo chiamata

- **`Rate.getCallCost`**
  - **Input**: gli stessi di prima (*per forza! è una delega!*), ovvero
    - istanti di inizio/fine chiamata
    - numero chiamato
  - **Come agire:**
    - Calcolare il **numero di intervalli** di cui è composta la durata della chiamata
      - per farlo: calcolare prima la durata della chiamata (= differenza in millisecondi fra data/ora di inizio e fine chiamata), poi dividerla per la durata dell'intervallo
    - Trovare la banda che fornisce il **costo per l'istante di inizio** della chiamata (**`Band.isInBand`**): *assumeremo valido quel costo per l'intera chiamata*
      - metodo privato **`getCostPerInterval(LocalDateTime)`** : cerca la banda applicabile al **`LocalDateTime`** passato e ne restituisce il costo per intervallo
    - Restituire il **prodotto** fra **numero di intervalli** e il **costo dell'intervallo**

# Algoritmo 1 (c)
# Diagramma di Sequenza



# Algoritmo 2
# Verifica di validità del piano

- **`PhonePlan.isValid`**
  - verifica che tutte le **`Rate`** siano valide
- **`Rate.isValid`**
  - verifica che tutte le **`Band`** [di una singola **`Rate`**] siano valide
  - verifica inoltre che ogni giorno della settimana abbia le 24 ore coperte

Farlo in blocco è improponibile

Occorre suddividere il problema in parti "affrontabili":

- **OCCHIO al problema della mezzanotte!**
- per verificare la copertura delle 24 ore NON possiamo agire come in MyCalendar, con intervalli chiusi a sinistra e aperti a destra!

# Il problema della mezzanotte

- Perché non lasciare l'intervallo aperto a destra, come nel calendario appuntamenti?
  - perché l'ultima **`Band`** termina a mezzanotte, che *appartiene già al giorno successivo:* ***non si può scrivere*** **`LocalTime.of(24,0)`*!***
  - mezzanotte è **`LocalTime.of(0,0)`** ma ***del giorno dopo!***
- Quindi, un approccio con intervallo aperto a destra rende-rebbe *complicato* rappresentare l'ultima Band del giorno
  - trattamenti ad hoc complicherebbero il codice in modo assurdo, *rendendolo illeggibile e non manutenibile* → INACCETTABILE
  - la rappresentazione deve essere naturale e seguire la realtà
  - adotteremo perciò **intervalli *chiusi a sinistra e a destra* [t1,t2]**, in modo da poter chiudere all'istante **`LocalTime.of(23,59)`**

# Bande su bande

- Se l'intervallo è chiuso a destra, come esprimere la *condizione di adiacenza?*
  - occorre che ogni band *"si attacchi"* alla precedente *senza buchi*
  - se la band attuale è [t1, t2], la successiva dovrà essere [t2+1, t3]
  - ..ma **+1** «*cosa*»*?*
- **Modello tempo-continuo vs tempo-discreto**
  - anche se il tempo fluisce *con continuità,* la minima granularità con cui lo misuriamo sono i *nanosecondi*
    → ha senso interpretare quel "+1" come **+1 ns**
  - test di adiacenza fra bande:

    se band1=[start1,end1] e band2=[start2,end2],

    `end1.plusNanos(1).isEqual(start2)`

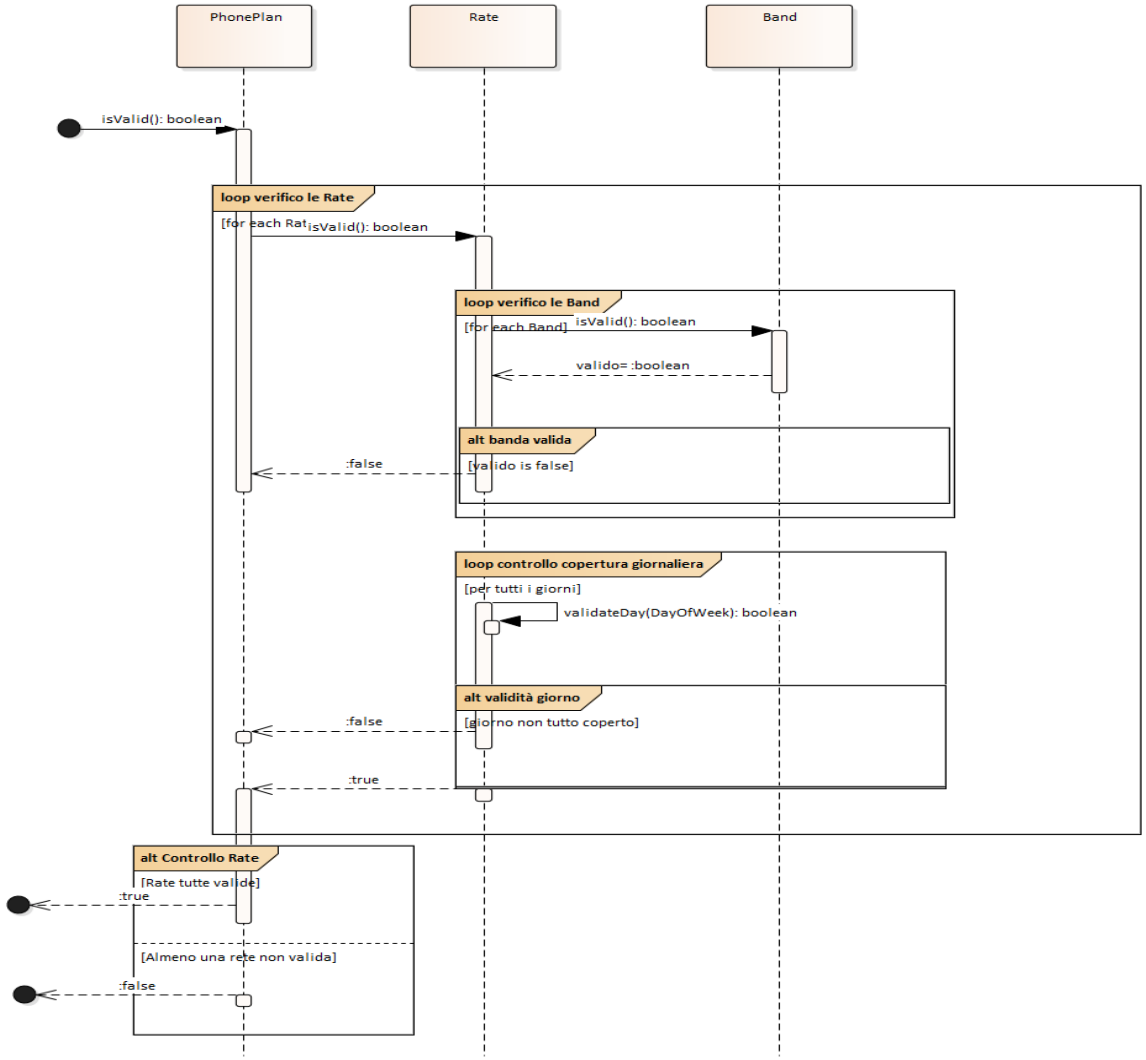
# Algoritmo 2
# suddivido il problema

- **`Rate.isValid`**
  - verifica che tutte le **`Band`** [di una singola **`Rate`**] siano valide
  - verifica inoltre che ogni giorno della settimana abbia le 24 ore coperte

Occorre suddividere il problema in parti "affrontabili":

- ogni banda deve essere valida → **`Band.isValid`**
- selezionare dall'insieme tutte e sole le **`Band`** del giorno corrente
- le **`Band`** selezionate devono *coprire le 24 ore*: per verificarlo,
  a. ordinare le **`Band`** selezionate in base all'ora di inizio
  b. verificare che la **prima `Band`** inizi alle **`LocalTime.MIN`** e l'**ultima** termini alle **`LocalTime.MAX`**
  c. verificare che **ogni altra `Band`** "intermedia" cominci *un nanosecondo dopo* la fine della precedente: **`end1.plusNanos(1).isEqual(start2)`**

# Algoritmo 2
# Diagramma di Sequenza

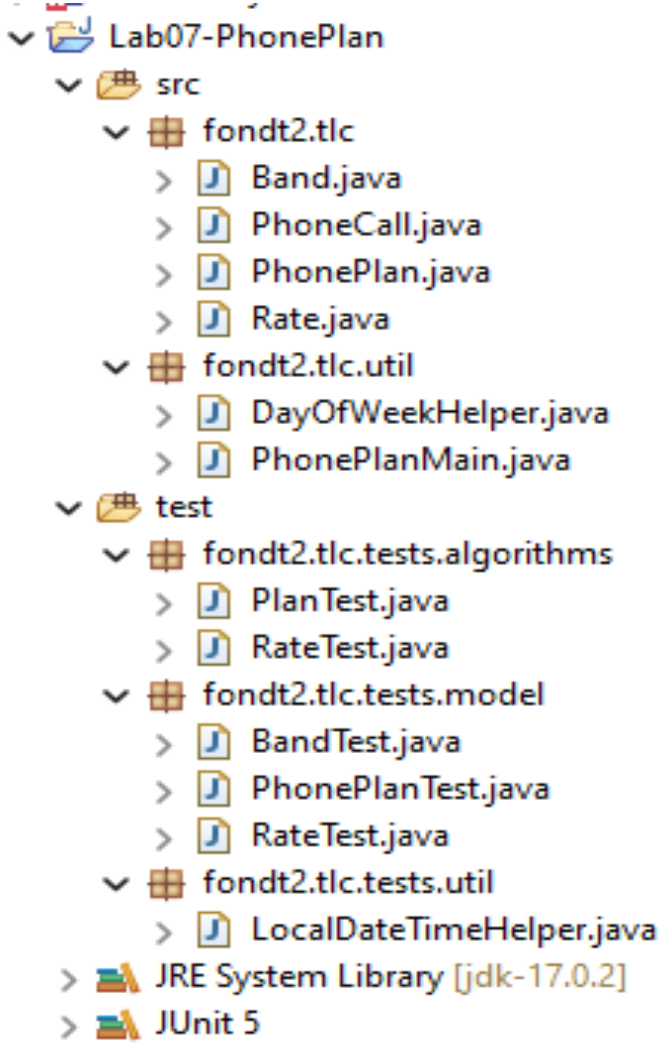


# Algoritmo 2
# Zoom inside **`validateDay()`**

- **`validateDay`** è un metodo privato di Rate che abbiamo utilizzato per scomporre il problema «la Tariffa è valida» e si occupa di capire se ogni singolo giorno è coperto 24h
- A sua volta **`validateDay`** utilizza altri metodi privati:
  - **`Band[] selectBandsInDay(DayOfWeek)`**
    estrae le **`Band`** coinvolte in quel dato giorno della settimana
  - **`void sortBandsByStartTime(Band[])`**
    ordina le **`Band`** nell'array in base all'ora di inizio
  - **`boolean validateBandsInDay(Band[])`**
    - prima, verifica che la prima **`Band`** dell'insieme ordinato cominci alle **`LocalTime.MIN`** e che l'ultima termini alle **`LocalTime.MAX`**
    - poi, verifica che ogni **`Band`** nell'elenco ordinato, tranne la prima e l'ultima, cominci un nanosecondo dopo la fine della **`Band`** precedente

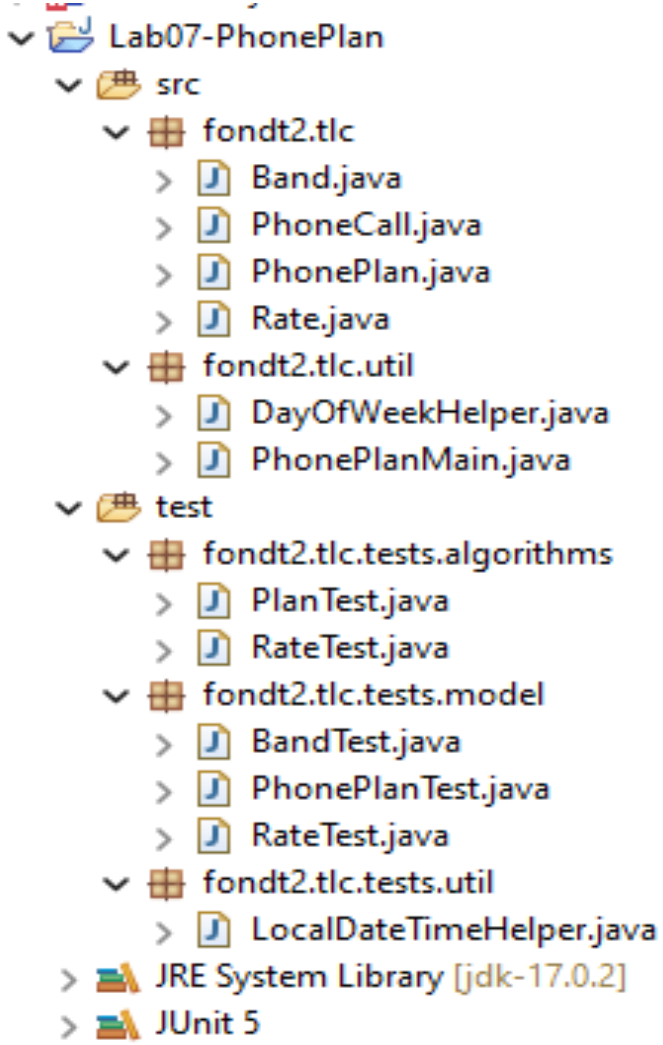
# Start Kit

- **Nello start kit ci sono:**
  - il progetto già fatto
  - la classe **`PhoneCall`**
  - Nel package **`util`** troverete **`DayOfWeekHelper`** e **`PhonePlanMain`** per fare un run dell'applicazione una volta terminato il codice
  - Test già pronti nel source folder «test»

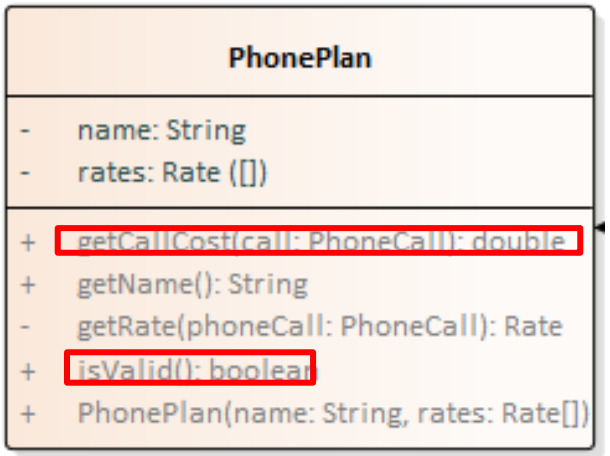


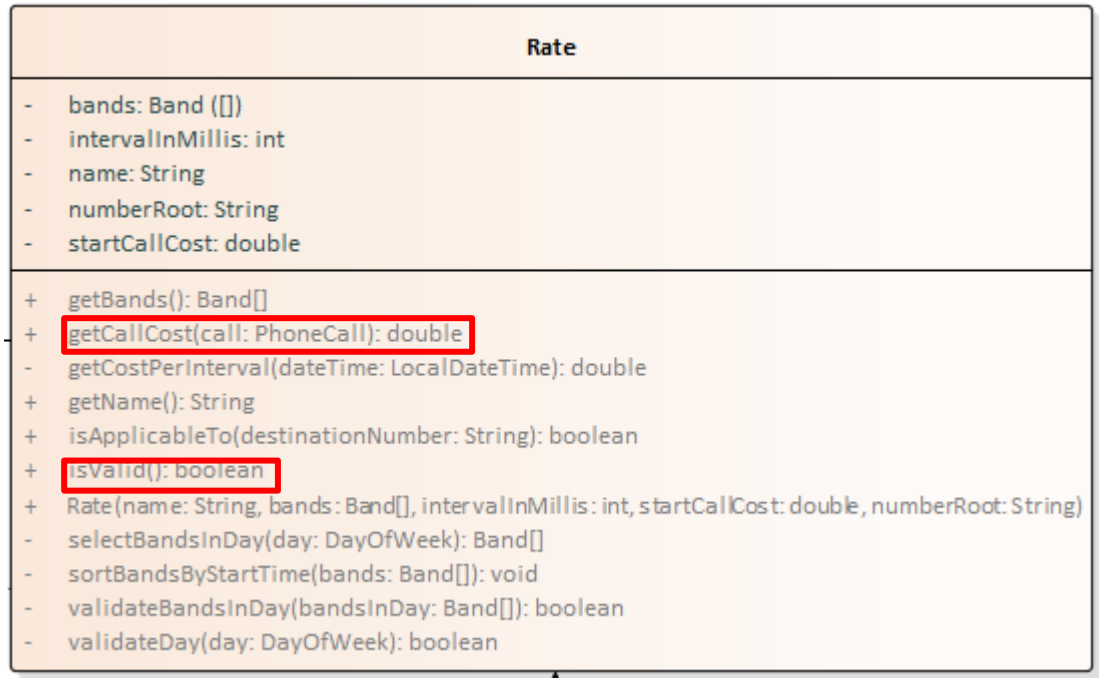
# Start Kit

- **Cosa dovete fare:**
  - Copiare le classi **`Band`**, **`Rate`** e **`PhonePlan`** che avete fatto la scorsa settimana nel progetto e sviluppare i metodi mancanti
- **Attenzione** ai nomi dei package.. altrimenti i test non funzioneranno!



# Rate e PhonePlan: completamento algoritmi



# Hey!

KEEP
CALM
AND
HAPPY
CODING