**FUNZIONALITA’**

1. Lanciando l’applicazione ci sono:
   1. Bottone per la gestione del calendario
   2. Bottone per la gestione delle camere
   3. Bottone per la gestione degli eventi
   4. Bottone di spegnimento del sistema
2. Con la gestione del calendario è possibile definire il calendario della struttura che ospita eventi e camere.
   1. La data iniziale è predefinita nella AppConfig = 01/01/2024
   2. La data finale può solo essere estesa
   3. Gli stati possibili per la struttura sono CHIUSO e DISPONIBILE
   4. Si può cambiare lo stato della singola data con il click dx del mouse
   5. Se una data è chiusa tutte le camere e le sale devono essere CHIUSO
   6. Se una data passa da DISPONIBILE a CHIUSO l’app deve fare un elenco delle camere e delle sale che devono essere rischedulate. Questo è gestito con un pattern Observer fra i server.service
3. La gestione delle camere è slegata da quella degli eventi.
   1. Riporta l’elenco delle camere
   2. Per ogni camera c’è l’elenco delle disponibilità
   3. Gli stati possibili sono DISPONIBILE, INUSO, CHIUSO e PULIZIA e PRENOTATA
   4. Se lo stato NON è CHIUSO, con il click di dx su può cambiare stato
   5. Lo stato CHIUSO è escluso dalle possibili scelte
   6. È possibile passare liberamente da uno stato all’altro, non c’è una sequenza predefinita. Lo stato PULIZIA va inteso (per un eventuale ampliamento) come MANUTENZIONE (per video, ecc). Il cambio di stato avviene attraverso un visitor con un generico.
   7. L’anagrafica delle camere è fissa, la sua gestione non è implementata (modifica prezzo, numero di letti e tipo). C’è un visitor che rende il prezzo ma non è usato (anche se funziona)
   8. Il nome della camera è gestito con il singleton
4. La gestione degli eventi è in realtà una maschera di eventi con una sottomaschera delle sale
   1. Il nome degli eventi è gestito con lo stesso singleton con un generico T che battezza sale, camere e calendari (potrebbero essere più di uno)
   2. La modifica dei seguenti campi è libera, una volta fatta, bisogna salvare l’evento
      1. Nome Organizzatore
      2. Costo partecipazione
      3. Numero partecipanti previsti
      4. Il tipo di catering richiesto (combo-box)
      5. Ora inizio dell’evento
      6. Ora fine dell’evento
   3. La data è delimitata fra il min e il max definiti dal calendario (anche date CHIUSO)
   4. Gli speech/interventi sono una lista che si può cambiare come e quando si vuole. Sono l’unica cosa che si può ELIMINARE (singolarmente con l’apposito bottone)
   5. La sottomaschera delle sale mostra la disponibilità delle sale. Ha lo stesso codice della maschera delle camere, ma qui il click dx è disabilitato mentre è abilitato quello sx sulle sale PRENOTATA, l’app rende i dati dell’evento che l’ha impeganata
   6. Il cambio di data dell’evento libera la sala precedentemente prenotata
   7. Il cambio di sala viene visualizzato anche nella sotto-maschera delle sale
   8. I bottoni ai piedi della maschera consentono di scorrere gli eventi esistenti, di salvare le modifiche apportate all’evento visualizzato o di crearne uno nuovo.
   9. Le modifiche non salvate si perdono cambiando evento
5. Lo spegnimento del sistema avviene con una richiesta REST al server

Non sono state gestite situazioni critiche e/o eccezioni …

Architettura RIGOROSAMENTE MVC PASSIVO (il controller funge da postino in entrambe le direzioni)   
le REST sono sincrone, ma SpringBoot e JavaFX si gestiscono i thread come vogliono loro  
I test sono soprattutto sui server (service, controller) e sul visitor. Per la parte client era più comodo vedere direttamente il comportamento delle maschere.

Tutte le classi hanno una presentazione in JavaDoc (/\*\* …..\*/)

**STRUTTURA DEL CODICE**

LATO SERVER

* 4 classi per gli adattatori (Double, Integer, LocalDate e LocalTime)
* 6 classi per i controller (Event + 3 per le risorse (Calendar, Hall e Room) + System per lo spegninento + Exception
* 6 classi per i service: Abstract (da cui ereditano Event e le 3 Risorse) + Singleton per il salvataggio su json dei contatori
* Nel server ci sono le classi
  + AppConfig con i dati di configurazione
  + ServerApplication che contiene il main del server
  + Interfaccia Subscriver per le notifiche del cambio di calendario della struttura
* nel model ci sono:
  + 4 classi Enum (CateringType, RoomType, ResourceType e State)
  + 3 classi + 1 interfaccia per i visitor
  + 1 classe per il Singleton
  + Classi di servizio (StateDate, Speech, EventInfo)
  + 1 interfaccia HasName
  + Event
  + La superClasse Resource e le 3 sottoClassi Calendar, Room e Hall

LATO CLIENT

* La classe ReceptinistApplication che contiene il main del client
* Le finestre
  + 2 classi di servizio DinamiCol e StaticCol
  + 1 classe per il menu
  + La WResource con <T extends Resources>. Per ridurre la complessità della classe ed aumentarne la coerenza è stato necessario suddividere il codice in 3 pezzi:
    - La gestione delle richieste REST (WResourceRest)
    - La gestione dei click sulle celle (WResourceClick)
  + WRoom, WHall e WCalendar ereditano e specializzano la WResource
  + WEvent è la maschera degli eventi, anche qui ho dovuto suddividere il codice; in WEventLayout c’è quello per la definizione del layout, in WEventRest quello per la gestione delle REST di questa classe

TEST DINAMICO eseguito con JUnit5

* 6 classi per i test sui server, coprono comunque molte funzionalità perché il codice dei server.service è in gran parte in AbstractService e una volta testato per una risorsa, funziona anche per le altre.
* 1 classe per il test sul visitor
* 1 classe per il test lato client

TEST STATICO eseguito SonarLint in locale … finchè non ho esaurito il servizio free

TEST STATICO eseguito con CodeMR … finchè non ho esaurito il servizio free