05/07/2020

PIAZZA_SPRINT 3

Sprint Goal

Essere in grado di gestire più clienti contemporaneamente.

Requisiti

-I requisiti sono gli stessi elencati nel file TFBO20ISS.pdf con **alcune assunzioni**.

-Assunzioni:
1) Si rilassa il vincolo de "un solo cliente in sala". Ora possono arrivare tutte le richieste di ingresso che si vuole. Come da requisiti, la stanza potrà accogliere fino a N=2 clienti contemporaneamente.
2) Il barman **non** è in grado di lavorare in parallelo a più ordini. Le preparazioni, supposte di breve durata, sono quindi sequenziali.
3) Il tempo di preparazione di un ordine è sempre lo stesso, a prescindere da cosa è stato ordinato.
4) I task del waiter **non** sono interrompibili.

Analisi dei Requisiti

COSA deve essere il sistema? Come mi aspetto che si comporti al termine di questo Sprint il sistema?

Dai requisiti è evidente come la decisione di accettare o meno una richiesta di ingresso dipenda dalla situazione attuale dei tavoli. In questa tabella che ho realizzato vengono mostrati i 6 possibili scenari significativi in cui possono trovarsi i due teatable nel momento in cui arriva una richiesta.

teatable2 / **teatable1**	tableclean	busy	dirty
tableclean	1	2	3
busy	2	4	5
dirty	3	5	6

Scenario 1-2-3
Arriva una richiesta di ingresso → Il waiter la prende in consegna il prima possibile[1] e accoglie il cliente.
Scenario 4
Arriva una richiesta di ingresso al waiter → Il waiter esegue il task *inform.*
Scenario 5-6
Arriva una richiesta di ingresso al waiter → Il waiter dovrà accettare lo stesso la richiesta di ingresso, ma dovrà pulire il tavolo prima di accoglierlo.

[1] dal momento che i task del waiter non sono interrompibili se il waiter è nello stato *serving_client* oppure *cleaning* porta a termine ciò che sta facendo. Solo dopo valuterà le nuove richieste, in ordine di arrivo. Se si trova in stato *rest* invece processerà la prima richiesta estraendola dalla coda delle richieste.

Bozze di Test Plan

Simulare gli scenari enunciati sopra e testare il comportamento del robot. (Si noti che lo scenario 1 è già stato testato nello Sprint 2). Segue un esempio di come può essere impostato il test di uno scenario:

```
@kotlinx.coroutines.ObsoleteCoroutinesApi
@kotlinx.coroutines.ExperimentalCoroutinesApi
    suspend fun testScenario4(){
        println("========== testScenario4 (2 table busy)========== ")
        delay(4000)     //Time to set up
        //TODO: set teatable(1, busy(1)) and teatable(2,busy(2)) on tearoomkb.pl

        // MANDO UNA RICHIESTA DI INGRESSO AL WAITERLOGIC
        requestToWaiterLogic("enterRequest", "enterRequest(3)")
        delay(3000)

        /*---Verrà eseguito il task inform---*/
        // CONTROLLO LO STATO DEL WAITERLOGIC
        checkResourceWaiterLogic("serving_client(3)")
        //... E CHE LA SUA POSIZIONE NON CAMBI, supposto si trovi in pos(0,0)
        for(i in 0..5){
            assertTrue(itunibo.planner.plannerUtil.atPos(0,0))
            delay(1000)
        }
    }
```

Analisi del Problema

PROBLEMATICHE RIPRESE DALL'OVERVIEW INIZIALE E DA AFFRONTARE IN QUESTO SPRINT

"the waiter should reduce as much as possible the waiting time of the requests coming from each client" : Nello Sprint 2, una volta terminato il task *convoyToExit,* il waiter si metteva subito a pulire il tavolo appena usato e solo dopo tornava ad ascoltare altre richieste di ingresso. In un'ottica di ridurre il tempo di attesa delle richieste dei clienti, come richiesto dal requisito, ora tornerà sin da subito in ascolto di richieste.

Da questo derivano due osservazioni:

1. Se siamo nello scenario 5 o 6 e arriva una richiesta di ingresso, il waiter dovrà eseguire una sorta di task "accept but clean first" : risponde **subito** e **in modo affermativo** al cliente che desidera entrare, in modo che questo riceva subito un feedback. Poi però, pulirà un tavolo[2] prima di raggiungere l'ingresso ed accogliere il cliente.
   ATTENZIONE! In questo caso, mentre il waiter pulisce il tavolo, si trova nello stato logico *serving_client* e non *cleaning* in quanto l'azione di pulizia è subordinata alla richiesta del cliente.

2. Il waiter non è più obbligato a pulire i tavoli al termine del task *convoyToExit.* Se si limitasse a pulire i tavoli solo nel caso specificato al punto precedente, ritarderebbe sempre l'accoglienza dei clienti → è bene che, se si trova nello stato logico *rest(X,Y)* e per un po' non riceve richieste, controlli se ci sono dei tavoli dirty . In caso affermativo il waiter andrà a pulirne uno e solo dopo tornerà in ascolto di richieste.
   Ovviamente **finché ci sono dei tavoli da dover pulire, il waiter non potrà eseguire il task *rest***.

   Attesa dovuta al task clean-caso peggiore: il waiter decide di pulire il tavolo poiché fino a quel momento non aveva richieste da servire e appena entra nel task *clean* gli arriva una richiesta. Questa richiesta deve aspettare $T\ clean$ prima di essere presa in consegna. → Dopo averne discusso con il committente è emerso che il task clean può essere scomposto in 3 sottotask ciascuno della durata di $\frac{T\ clean}{3}$.

   Questi nuovi task li chiamiamo:
   - clean1 (consisterebbe nello sparecchiare).
   - clean2 (consisterebbe nel sanificare).
   - clean3 (consisterebbe nell'apparecchiare).

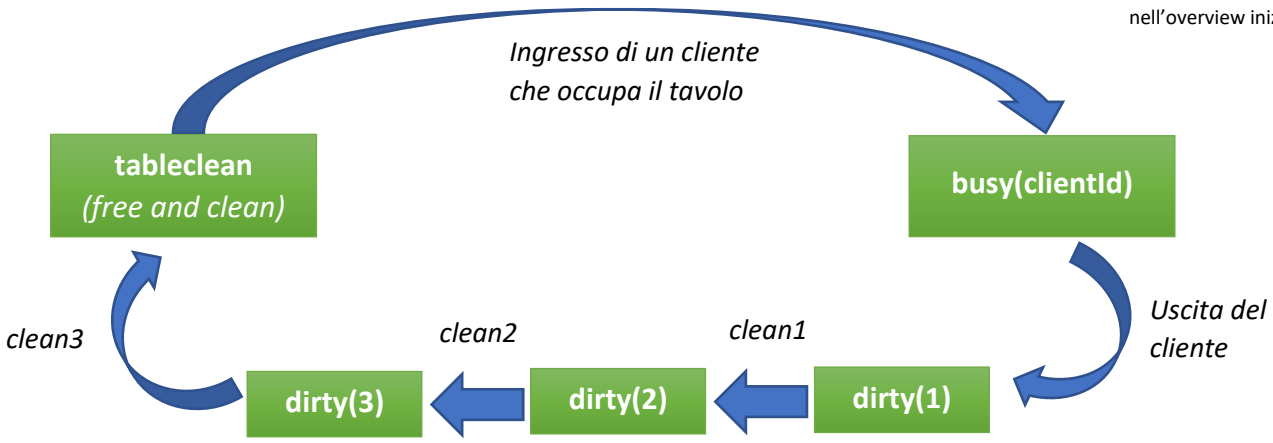
Se dopo ogni sottotask facciamo controllare al waiter se ci sono nuove richieste e nel caso gliele facciamo prendere in consegna, il caso peggiore di attesa dovuta al task clean si riduce a $\frac{T\ clean}{3}$. Ovviamente sarà compito del waiter ricordarsi lo stato di pulizia raggiunto per ogni tavolo, in modo da riprendere a pulirli da dove era rimasto.

[2] Alla luce della divisione del task clean in 3 sotto task il waiter dovrà scegliere di pulire il tavolo che tra i due è nello stato di pulizia più avanzato, in modo da far attendere meno il cliente.

Stato dei teatable : Ecco come risulta alla luce di ciò la transizione di stato dei tavoli.

UPDATED

*cambiato rispetto all'analisi fatta nell'overview iniziale.



Countdown del maxStayTime : Avendo rilassato il vincolo di 'un solo cliente in sala' **nasce una nuova problematica che nello scorso Sprint non avevamo**. Nel caso in cui un cliente dica "Sono pronto a ordinare", oppure "sono pronto a pagare" ma il waiter sta servendo un altro cliente o sta pulendo un tavolo, **potrebbe scadere il maxStayTime mentre il waiter sta finendo di compiere la propria attività**: e il cliente in tutto ciò non ha colpa.
Essendo una problematica legata alla 'simulazione dei clienti' si demanda una possibile soluzione alla fase di Progetto.

WebGUI: Nell'ottica di avere un feedback più chiaro ed efficiente con il committente, il progettista valuti l'opportunità di realizzare un'interfaccia grafica con alcuni pulsanti che possano permettere di "suonare il campanello" e di simulare l'interazione dei due clienti che possono essere presenti in sala. Nel realizzarla è bene tenere presente di permettere solo un'interazione utente adeguata (es: se non c'è nessun cliente al Tavolo 1 il bottone readyToOrder dovrà essere disattivato.).

Si segnala al progettista di realizzare il tutto come Web GUI. Questa, infatti, essendo fruibile da ogni tipo dispositivo dotato di browser (smartphone, pc, tablet ecc…) è un'opportunità che da maggior valore al prodotto rendendolo più appetibile e facilmente vendibile sul mercato.

Inoltre, questa interfaccia potrà servire anche:
-a fini di testing.
-a soddisfare il requisito della currentSituation della tearoom che affronterò nello Sprint4.

ARCHITETTURA LOGICA

L'architettura logica rimane invariata rispetto a quella riportata nello Sprint 2.

MODELLO ESEGUIBILE

Si veda a questo link il modello eseguibile ProblemAnalysisModel_Sprint3.qak. Si tratta di una revisione del modello eseguibile prodotto nello Sprint 2.

Test Plan

Si veda il codice completo del Test Plan al seguente link: testSprint3.kt

Progetto

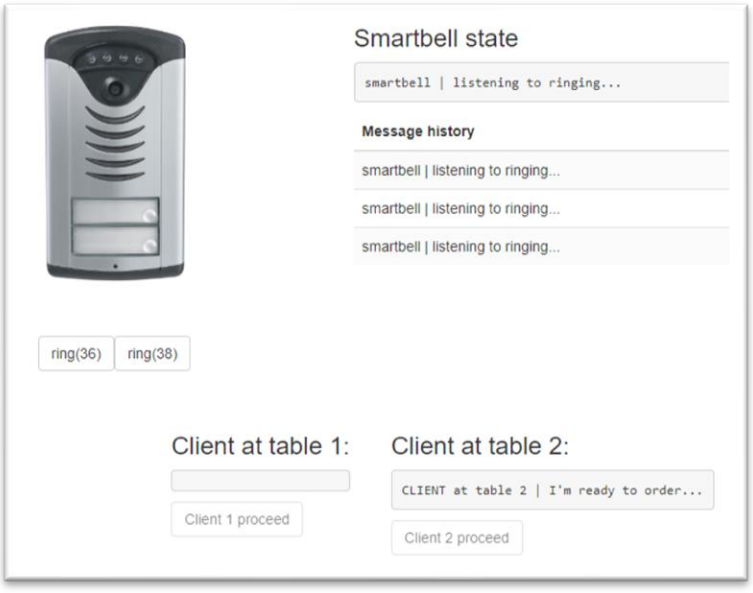
Alcune note:

- tearoomkb.pl:
  Implementata numbusytables(N), rule Prolog che permette di contare quanti tavoli sono in stato busy(CID).

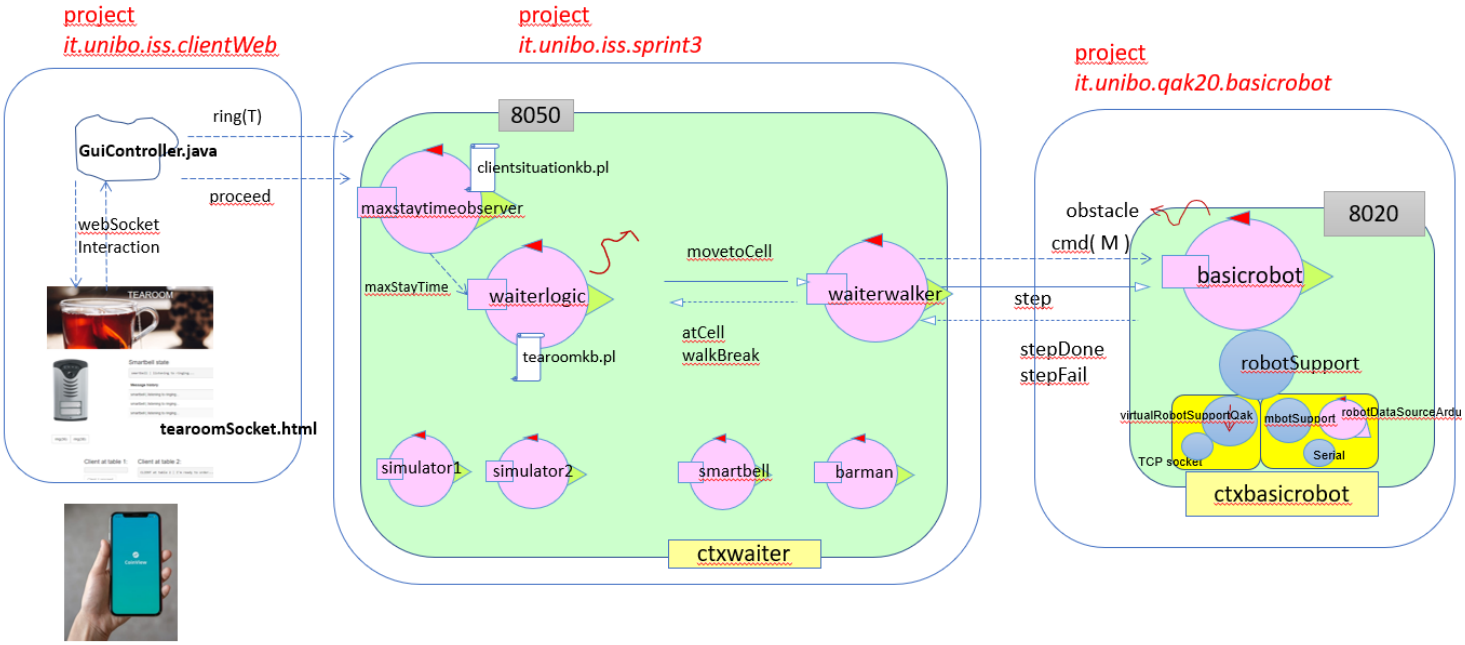
- Il modello presenta non più uno solo bensì due client_simulator: client_simulator1 e client_simulator2 simulano rispettivamente il cliente seduto al tavolo 1 e al tavolo 2.

- countdown maxStayTime : Adesso sono i client simulator ad emettere gli eventi local_preparation e local_leaving in modo che il maxStayTimeObserver possa stoppare immediatamente il timer.

- La web Application è stata realizzata utilizzando il framework SpringBoot.
  Nella webGUI sono presenti i pulsanti per simulare il suonare il campanello e per simulare il comportamento dei clienti al tavolo.
  L'interazione tra pulsanti e Controller avviene tramite WebSocket.
  Gli stati dei clienti e della smartbell vengono osservati sfruttando il fatto che i QActor sono risorse COAP.
  Nella webGUI vengono mostrati e aggiornati automaticamente quando cambiano .



Il progetto complessivo, a seguito di questo Sprint risulta essere il seguente:



[1] I messaggi di interazione tra barman-waiterlogic, smartbell-waiterlogic e simulatorN-waiterlogic **non** sono stati qui riportati per motivi di leggibilità. Per vederli si rimanda all'architettura logica riportata nell'Overview Iniziale. Al momento sono rimasti invariati.

[2] Si noti che nel progetto it.unibo.iss.sprint3 tutti gli attori sono stati messi nello stesso contesto per puro motivo di semplicità di esecuzione.

SPRINT 3 – REVIEW