28/06/2020

PIAZZA_SPRINT 2

Sprint Goal

Essere in grado di gestire un cliente alla volta.

Requisiti

-I requisiti sono gli stessi elencati nel file TFBO20ISS.pdf ai quali aggiungiamo **alcune assunzioni semplificative**.

-Assunzioni:

In aggiunta alle due assunzioni fatte nello Sprint_1:
1) Si suppone che non possa arrivare una seconda richiesta di ingresso fintanto che il cliente precedente non è stato accompagnato all'uscita.
2) Al barman, di conseguenza, verrà chiesto di lavorare alla preparazione di un ordine alla volta.
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? Mi aspetto che al termine di questo Sprint il sistema si comporti in questo modo:

Scenario 1 "Arriva un cliente con T°>=37.5": La smartbell è in grado di respingerlo autonomamente senza assegnarli alcun Client ID. Il waiter non viene notificato di nulla.

Scenario 2 "Arriva un cliente con T°<37.5": La smartbell inoltra la richiesta al waiter. Il waiter controlla la situazione tavoli e se questa è favorevole esegue in sequenza i task *accept, reach, convoyToTable* (al termine del quale fa partire il timer maxStayTime). A questo punto aspetta un messaggio dal cliente che è pronto ad ordinare, quindi esegue il task *order* e trasmette l'ordine al barman. Quando il barman ha ultimato la preparazione lo comunica al waiter che esegue il task *serve*. A questo punto si presentano due sottoscenari:

> **2.1**: il waiter aspetta un messaggio dal cliente che gli dice di essere pronto a pagare. In questo caso, il waiter esegue in sequenza i task *collect*, *convoyToExit* e *clean*.
>
> **2.2**: Se dovesse scadere invece il maxStayTime per il cliente, sarà compito del waiter accertarsi se è opportuno riscuotere il pagamento del cliente oppure accompagnarlo all'uscita senza farlo pagare.

Dal momento che i task del waiter non sono interrompibili, solo una volta ultimato il task *clean* il waiter è pronto a valutare una nuova richiesta di ingresso di un cliente.

BOZZE DI TEST PLAN

Possiamo pensare di testare i vari task del waiter come enunciato nel documento di overview. Si noti che sotto le assunzioni fatte in questo Sprint, il task *inform* non verrà mai eseguito.

```
@Test
	fun testwaiterLogic(){
		runBlocking{
			while( waiterLogic == null || waiterWalker == null || smartbell == null ){
				delay(initDelayTime)  //time for system to start
				waiterWalker = it.unibo.kactor.sysUtil.getActor("waiterwalker")

				waiterLogic  = it.unibo.kactor.sysUtil.getActor("waiterlogic")
				smartbell    = it.unibo.kactor.sysUtil.getActor("smartbell")

			}
			//"HIT THE SMARTBELL" to simulate a client enter-request
			forwardToSmartbell("ring", "ring(36.8)" )

			testAccept()
			testReach()
			testConvoyToTable()
			testTake()
			testServe()
			testCollect()
			testConvoyToExit()
			testClean()
			testRest()
		}
		println("testWaiterLogic BYE")
	}
```

Si possono testare anche gli scenari 1, 2.1. e 2.2 sopra citati.

Analisi del Problema

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

Il waiter, o qualcuno per lui, deve tenere traccia della situazione corrente della TeaRoom : con riferimento a questa problematica evidenziata nell'overview possiamo dire che il problema richieda di rappresentare opportunamente la conoscenza riguardo al mondo della stanza. A questo scopo si segnala al progettista la possibilità di scrivere una base di conoscenza **tearoomKB** usando un paradigma di programmazione dichiarativo, che ben si adatta a questo tipo di problemi.

Al momento possiamo dire che nella base di conoscenza ci interessano le seguenti informazioni:

- posizioni degli elementi significativi della tea Room (vedi Sprint 1).

-stato dei tavoli (vedi problematica Stato dei teatable nella overview).

-stato del waiter (vedi problematica Stato logico del waiter nella overview):

-rest (X,Y) [1]
-serving_client (CID)
-cleaning (table(N))

[1] Ho fatto confluire i due stati logici rest e doing_nothing in un unico stato rest(X,Y) che indichi anche dove si trova il waiter.

Tenendo presente il motto affrontato a lezione:

> Do not communicate by sharing memory; instead, share memory by communicating

vogliamo evitare di condividere una singola base di conoscenza tra più attori.
Possiamo pensare di assegnare la proprietà della base di conoscenza al componente del waiter che modellerà la business logic in quanto al momento è l'unico componente che deve consultarla e che deve aggiornarla.

Momento in cui occupare un tavolo (si vedano le considerazioni su questa problematica nell'Overview).

Countdown del maxStayTime (si vedano le considerazioni su questa problematica nell'Overview). Inoltre: dal momento che il waiter deve dedicarsi ai clienti ed essere disponibile a rispondere alle loro richieste, se si dedicasse ad osservare il tempo residuo per ogni cliente nessuna delle due attività verrebbe svolta bene.

Questa problematica richiede la **presenza di un'entità dedicata ad osservare la situazione del maxStayTime per i clienti in sala**. Il comportamento di quest'entità deve essere tuttavia subordinato al waiter in quanto è il waiter a conoscere i momenti esatti in cui un timer va fatto partire, interrotto e fatto riprendere. **Sarà quindi un'entità *slave.***

Ci chiediamo: questa entità ha la responsabilità di osservare il tempo di permanenza di un solo cliente o di tutti quelli in sala? Al momento, essendo il numero di clienti massimi in sala pari a 2, si ritiene non oneroso far controllare entrambi dalla stessa entità.

Sincronizzare il comportamento del cliente con quello del waiter :

Dal momento che da requisiti ***non è richiesto di implementare un'applicazione per il cliente*** abbandoniamo la soluzione del modello a stati introdotta nell'overview e, per stare sul semplice, lo simuliamo → Possiamo immaginare il cliente come un semplice pacco da prendere, portare al tavolo ecc.. il controllo della situazione ce l'ha il waiter. Tuttavia, la "palla" passa al cliente nel momento in cui deve consultare il menù o deve consumare il tea.

Revisione del messaggio ring :

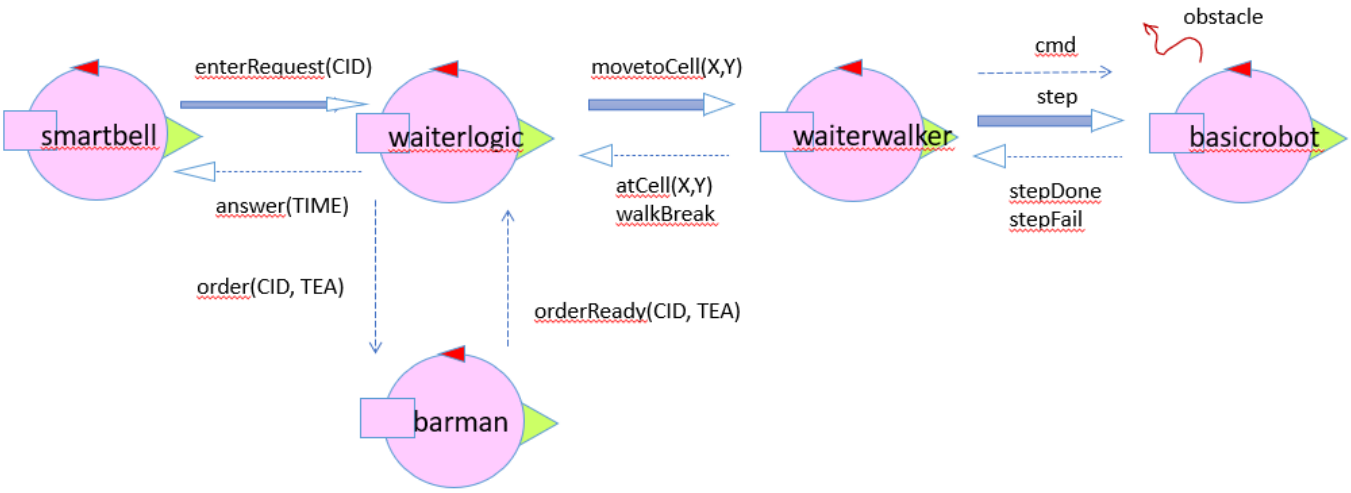
Anche l'interazione smartbell-client viene rimossa e simulata con degli auto-msg ring che si manda da sola la smartbell. Dal momento che anche la rilevazione della temperatura è simulata, inseriamo la temperatura corporea del cliente che suona il campanello come Payload del messaggio ring. Eliminiamo poi i due messaggi reply `waiterResponse` e `refused` e trasformiamo ring in un Dispatch.

```
/*-------- Auto-msg smartbell -------- */
Dispatch ring        : ring(TEMPERATURE)   //Per scopi di simulazione
```

ARCHITETTURA LOGICA

Perseguendo la stessa strada intrapresa nello Sprint_1 e nell'ottica di comporre "servizi riusabili", possiamo pensare al waiter come un'entità che vista dall'esterno appare monolitica, ma che al suo interno è composta da più attori che collaborano e hanno ciascuno determinate responsabilità.

Se nello Sprint_1 avevamo diviso il waiter in mente (**waiterwalker**) e corpo (**basicrobot**), possiamo pensare di aggiungere un ulteriore livello di mente, responsabile della business logic del waiter: lo chiamiamo **waiterlogic** ed è la vera mente del waiter. Conosce le informazioni riguardo al dominio (cioè riguardo alla teaRoom e al suo stato) e ha la possibilità di richiedere ad un attore subordinato, l'attore waiterwalker.qak, di volersi spostare in una determinata cella.



MODELLO ESEGUIBILE

Si veda ProblemAnalysisModelSprint_2.qak nel progetto *Sprint2/it.unibo.iss.sprint2/documents.*

Test Plan

Si veda testWaiterLogic.kt

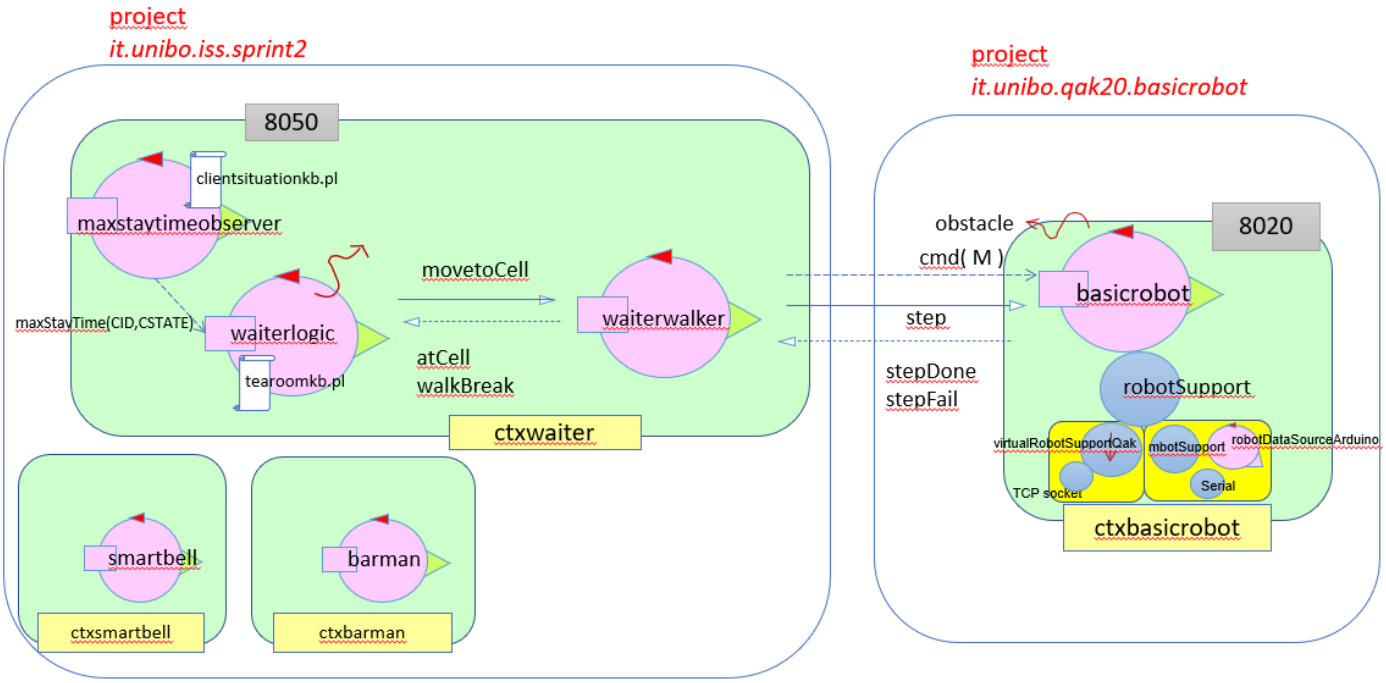
Progetto

NOTE:

-tearoomkb.pl : La base di conoscenza relativa alla Tea Room è stata realizzata usando il linguaggio TuProlog.

-Countdown del maxStayTime :

- Ho aggiunto l'attore maxStayTimeObserver (maxStayTimeObserver.qak) che viene informato dalla waiterlogic ogni volta che un cliente inizia a consultare il menù, trasmette l'ordine al barman o inizia a consumare il tea. Questo attore mantiene aggiornata una propria base di conoscenza riguardante la situazione dei clienti in sala (clientsituationkb.pl) e, ogni secondo, controlla se per qualcuno dei clienti è scaduto il tempo. In tal caso notifica il waiterlogic, informandolo anche su cosa stava facendo il cliente quando è scattato il timeout ("consulting" oppure "consuming") in modo che il waiter possa capire se deve far pagare il cliente o no.
- L'interazione waiterlogic → maxstaytimeobserver è stata modellata con degli eventi locali.
  In questo modo:
  -Il sistema è già predisposto alla presenza di più observer.
  -Un domani le informazioni che vengono emesse possono interessare anche altri slave della waiterlogic appartenenti allo stesso suo contesto.
  -Non ho problemi di privacy/sicurezza perché un evento locale rimane dentro al contesto.



[1] I messaggi di interazione tra barman-waiterlogic e smartbell-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.

SPRINT 2 – REVIEW

Si valuti la possibilità di mettere delle ReadLine o dei punti di stop nel sistema in modo che in quei punti il controllo passi a me programmatore. In questo modo sarà più facile debuggare il sistema e far comprendere il suo funzionamento al committente.