13/07/2020

PIAZZA_SPRINT 4

Sprint Goal

Essere in grado di visualizzare la current situation della tearoom nella Web GUI.

Requisiti

-I requisiti sono gli stessi elencati nel file TFBO20ISS.pdf con **le stesse assunzioni fatte nello Sprint 3**.

-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 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

Il requisito da soddisfare è il seguente:

> As a ***manager***:
> I intend to be able to see the ***current state*** of the ***tearoom*** by using a browser connected to a web-server associated to the application.

Definizione di current situation : come emerso dal documento di overview iniziale, lo stato corrente della stanza che interessa al manager si compone delle seguenti informazioni:

current situation = stato del waiter & stato dei tavoli.

L'aggiunta di ulteriori informazioni, se verranno richieste dal committente, viene demandata a sviluppi futuri.

Bozze di Test Plan

Segue una bozza di Test Plan che riporta un particolare scenario d'esecuzione. Nel test plan questa bozza verrà formalizzata e sarà così possibile mostrare al committente una simulazione automatizzata del sistema.

```
testSickRing()              // test di una richiesta di cliente con febbre.
test2ClientsEnterRequest()  // test sull'arrivo di due clienti in sequenza, uno dopo l'altro.
testClientSimulation()      // i due clienti ordinano
testBusyRoom()              // test di una richiesta fatta con la stanza piena.
testPaymentAndMaxStayTime() // il cliente 1 paga ed esce, il cliente 2 fa scadere maxStayTime ed esce.
testCleanTable()            // il waiter pulisce i tavoli.
testRest()                  // il waiter torna alla home a riposare.
```

Analisi del Problema

PROBLEMATICHE

Stato della tearoom centralizzato : Lo stato corrente della stanza è bene che sia centralizzato per 2 motivi:
-Per far fronte al requisito del manager.
-Per facilitare il testing che quindi può essere impostato e ruotare tutto intorno ad esso.
Come era emerso anche in Sprint precedenti, l'attore waiterlogic è il responsabile dell'aggiornamento dello stato della stanza e lo tiene memorizzato nella propria base di conoscenza *tearoomkb.pl*.

Visualizzare la current situation sulla webGUI : le informazioni relative allo stato della stanza si trovano lato server (progetto *it.unibo.iss.sprint4*) e io devo visualizzarle nella GUI. Nasce quindi una problematica:

Il problema richiede che le informazioni vengano trasmesse dal server alla GUI

- **chi** ha il compito di trasmetterle? È la web Application a dover prendere l'iniziativa facendo richieste al server, oppure è il server a dover prendere l'iniziativa e inviare le informazioni di stato alla webGUI?
- **quando** è opportuno trasmetterle?

Dal momento che da requisiti si chiede di visualizzare la ***current*** situation, emerge la necessità di visualizzare lo stato della stanza sempre aggiornato → sarà necessario trasmettere il nuovo stato alla GUI ogni volta che questo cambia.
Si demanda al progettista la scelta di una soluzione in linea con le analisi appena fatte.

Nello SPRINT 3 era già emersa la necessità di realizzare una Web GUI, con lo scopo di rendere più proficuo il momento di confronto e feedback con il committente. Per dare la possibilità al committente (manager) di avere tutto sotto controllo nella stessa webGUI si segnala al progettista di utilizzare la stessa web GUI sviluppata nello sprint precedente, arricchendola dei dati necessari a soddisfare il requisito della *current situation*.

ARCHITETTURA LOGICA

L'architettura logica rimane invariata rispetto a quella adottata nello Sprint 3.

MODELLO ESEGUIBILE

Si veda a questo link il modello eseguibile ProblemAnalysisModel_Sprint4.qak.

Test Plan

Si veda il test plan completo a questo link: testSprint4.kt.
I test sono stati condotti controllando il *current state* della tearoom e confrontandolo con quello atteso situazione per situazione.

Progetto

1. **"…è necessario trasmettere alla GUI il nuovo stato ogni volta che questo cambia".**

(dall'analisi del Problema)

Alla luce di ciò è chiaro come né una soluzione in cui la webGUI interroga a polling il server, né una soluzione in cui il server manda informazioni alla webGUI a intervalli regolari siano adeguate.
Una soluzione basata sul pattern Observer è molto più adatta a risolvere questa problematica!

Se immaginiamo lo stato della stanza come Observable, sarà possibile agganciarle uno o più observer che recepiscano il cambiamento di stato e agiscano di conseguenza (facciano visualizzare lo stato sulla pagina).
Ulteriore vantaggio del pattern observer: L'entità Observable non sa a priori chi e quanti saranno gli observer che sottoscriveranno il proprio interesse (maggior disaccoppiamento tra le entità).

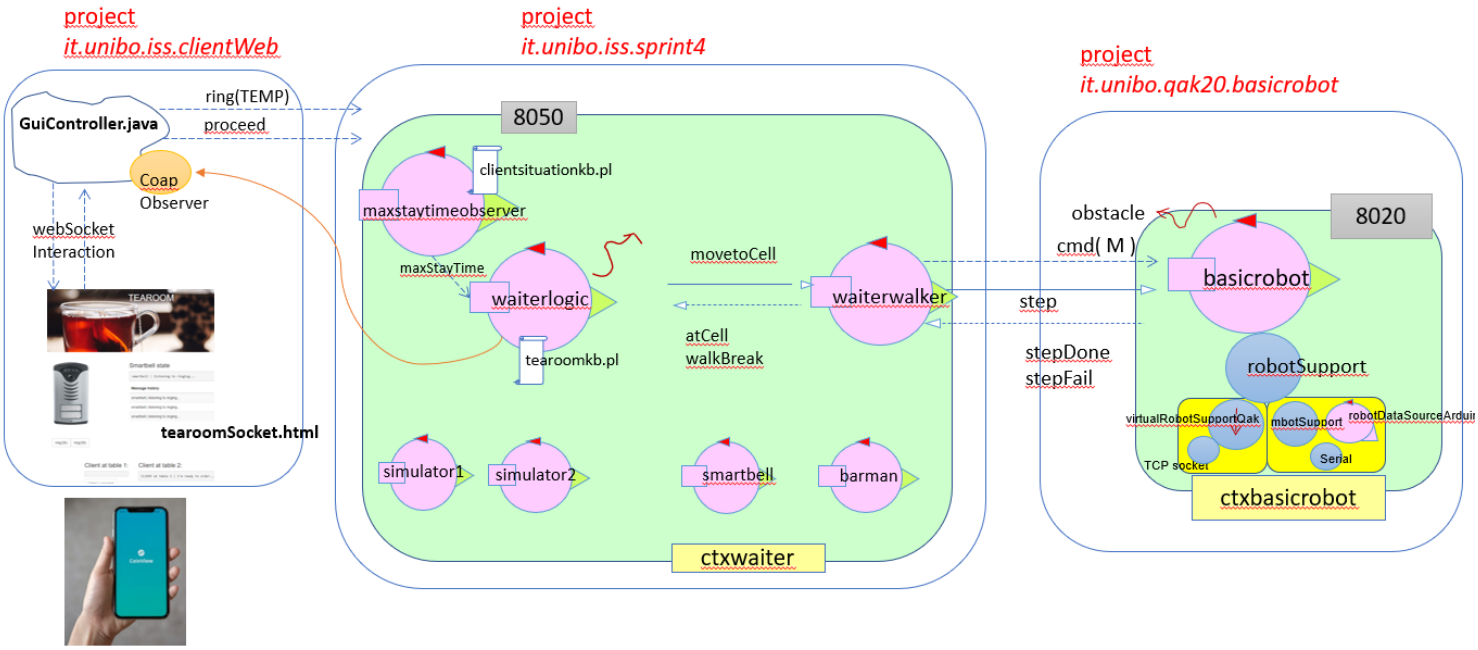
Nasce una ulteriore problematica:
-**observer locali oppure observer remoti?** Possiamo sfruttare il fatto che l'infrastruttura QakActor renda i QActor delle entità osservabili anche in remoto tramite protocollo Coap. Nel nostro caso lo stato si trova nel waiter a cui basterà fare un `updateResource[#"$nuovoStato"#]` ogni volta che lo stato cambia. Potremmo così utilizzare un Observer remoto locato sulla webApplication il quale, ogni volta che cambia lo stato, recepisce il cambiamento di stato.

2. single page application con webSocket oppure interazione paginaHtml ←→ GUIController REST?

Seguendo l'interazione classica di Spring Boot (basata su protocollo http) la pagina html dovrebbe richiedere esplicitamente l'aggiornamento delle informazioni. ☹

Sfruttando le webSocket possiamo implementare una single page application: la pagina html si sottoscrive ad una topic in cui il server può mandare quando vuole delle informazioni, senza il bisogno di essere sollecitato dalla pagina stessa 😊

Il coapObserver remoto (si trova lato webApp) quando rileva un cambiamento di stato lo deposita via WebSocket su una determinata topic sfruttando un componente Spring detto SimpMessagingTemplate. La pagina html si sarà registrata a quella topic e avrà agganciato una funzione di callback in grado di mostrare sulla pagina, ogni volta che arriva un nuovo messaggio, il messaggio ricevuto.



[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 semplicità di esecuzione.

[3] Per motivi di leggibilità è stata riportata solo l'observable relationship tra CoapObserver e QActor waiterlogic. La stessa relazione è però presente anche tra altri 3 observer che osservano rispettivamente i QActor simulator1, simulator2 e smartbell.

SPRINT 4 – RETROSPECTIVE

Towards a Exagonal Architecture

L'idea per un refactoring futuro può essere quella di andare verso un'architettura della nostra applicazione che non sia più layered come è stata impostata fino ad ora, bensì **esagonale**.

È il waiter che cambia lo stato in seguito ad una propria azione? Oppure è lo stato che viene sollecitato a cambiare e, in base al proprio cambiamento comanda le azioni del waiter (Visione domain centered-Modello Esagonale). Fino ad ora è stato adottato il primo approccio.

Dallo studio dell'applicazione fin qui condotto, emerge però come il domain Model sia centrale e determinante in ogni scelta che viene presa da waiter.
Possiamo pensare quindi di **mettere al centro il domain model** e, quando questo riceve degli eventi che ne inducono un cambiamento (es: messaggi con richieste di ingresso) potrà invocare determinate azioni sul waiter (che in questo modo diventa un vero e proprio servizio).

Ora le azioni del waiter sono subordinate ad un'invocazione richiesta dal domain Model che tenta di cambiare il proprio stato: trovandoci ad eseguire in un *Real World* se poi il servizio invocato andrà a buon fine lo stato cambierà altrimenti no.
Ora non è più il waiter che fa le azioni e comanda il cambiamento del domain Model. Si è ribaltata la visione.

One client in the hall: Alcune considerazioni

Il requisito opzionale one client in the hall comporta che, una volta fatta la ring della smartbell, la successiva non potrà essere fatta fino a quando il waiter non ha fisicamente accolto chi ha suonato.

Una ring non può essere effettuata se il waiter sta eseguendo un task convoyToExit. Viceversa, un convoyToExit non può essere fatto se è stata fatta una ring poiché significa che la hall è occupata da chi ha suonato.

Possibile soluzione (progetto): queste due azioni diventano due azioni che per eseguire devono richiedere **un lock sulla risorsa hall**, la quale andrà modellata come una risorsa ad accesso mutuamente esclusivo.