SPRINT 2

PROBLEM ANALYSIS

Assunzione sull'operazione di pulizia

In questo sprint si considera l'operazione di pulizia di un tavolo come atomica, cioè che non può essere interrotta.

Descrizione del sistema

Dai requisiti si dovrà creare un sistema composto dalle seguenti entità:

- **Waiter**: gestisce la sala da the interagendo con il cliente e gli altri attori.
- **Smartbell**: interagisce direttamente con il cliente e si occupa di notificarne l'arrivo al waiter.
- **Barman**: interagisce esclusivamente con il waiter.
- **Client (esterno):** simula un cliente ed il suo ciclo di vita nel sistema.
- **Manager (esterno):** deve avere accesso al web server per consultare lo stato della stanza

All'interno dello Sprint 1 si è già visto come realizzare i vari componenti sopra elencati, nello Sprint 2 invece sorge la necessità di **dover gestire più clienti alla volta**, per fare ciò è necessario:

- Modificare il waiter in modo tale che sia in grado di rispondere alle richieste della smartbell senza dover attendere la fine dell'attività in corso.
- Modificare la smartbell in modo tale che sia in grado di gestire più clienti insieme, compresi clienti che non hanno la temperatura adatta per entrare all'interno della sala.
- Risolvere la gestione dello stato dei tavoli.

Oltre a queste modifiche, sarà necessario soddisfare il requisito del web server per permettere al manager di tener sotto controllo lo stato della sala, e realizzare una GUI per poter testare il sistema creato, possibilmente anche da remoto.

Problematica del current state

Nel primo sprint si è gestito un solo tavolo, il cui stato viene salvato all'interno di una variabile del waiter.

Ricordiamo che lo stato di un tavolo può essere:

- **Pulito** (clean): è la situazione iniziale dei tavoli e quella successiva all'operazione di igienizzazione da parte del waiter.
- **Occupato** (taken): è la situazione in cui si trova il tavolo quando è presente un cliente seduto.

- **Sporco** (dirty): è la situazione in cui si trova il tavolo dopo che il cliente ha finito di consumare il tea e il waiter ancora non provvedere all'igienizzazione dello stesso.

La soluzione trovata nel primo sprint risulta essere inefficace in una situazione più complessa, perché obbligherebbe il waiter a tenere memoria di ogni singolo tavolo presente nella stanza, è necessario quindi approfondire le alternative indicate in precedenza.

Mantenimento del current state

Gli *approcci generali* per il mantenimento del current state sono due:

- **Centralizzato:** le informazioni vengono mantenute all'interno di una sola entità che quindi si fa carico di gestire le informazioni e fornirle quando richiesto. L'approccio richiede meno sforzo di coordinazione fra le varie entità.
- **Distribuito:** ogni entità si fa carico di mantenere la parte dello stato che la riguarda. Quando necessario, le informazioni vengono reperite tramite scambio di messaggi.

Riguardo l'approccio centralizzato è possibile scegliere un'entità già presente per mantenere lo stato del sistema. Ad esempio, una possibile soluzione è quella di delegare al waiter questa mansione. Identificate questi due approcci generali, si possono ottenere approcci alternativi ibridi.

Approccio Centralizzato

Come già detto, tale approccio prevede che lo stato di tutte le entità in gioco sia mantenuto in un unico punto.

Analizzando le richieste del committente si nota che il waiter è, intrinsecamente al suo ruolo, conscio dello stato di tutti gli altri componenti del sistema. Difatti:

- **Tavoli**: lo stato dei tavoli cambia soltanto per azione del waiter, il quale può far accomodare dei clienti, accompagnarli all'uscita al termine della consumazione ed igienizzare i tavoli rendendoli nuovamente fruibili.
- **Barman**: il barman non cambia mai di stato senza che il waiter o ne sia la causa scatenante oppure ne sia informato.

Alla luce di tali considerazioni, **l'approccio centralizzato risulta una scelta possibile** per il mantenimento del current state in quanto il waiter si configura come entità cardine del sistema, informata in ogni momento dello stato di ogni altra entità.

Approccio Distribuito

Tale approccio prevede una completa indipendenza, per quanto riguarda il mantenimento del current, tra le entità in gioco. Nello specifico, non si avrà un'unica entità responsabile dello stato complessivo del sistema, bensì **ogni entità sarà responsabile del mantenimento del proprio stato** e lo renderà disponibile quando richiesto.

Approccio scelto

Nonostante, come già evidenziato, il waiter si configuri come punto di snodo delle informazioni del sistema, l'approccio completamente centralizzato presenta degli svantaggi.

Nello specifico, le considerazioni fatte a proposito del waiter vengono meno ad un cambiamento, per certi versi probabile, del numero di camerieri che gestiscono la stanza da tea. Questa eventualità, non remota, renderebbe del tutto inadeguato il prodotto che si vuole sviluppare.

Avendo ben a mente questa criticità, si opta per un approccio intermedio: ogni attore mantiene il proprio stato e lo rende disponibile, ma la gestione dei tavoli è delegata ad un'unica entità, chiamata **tearoom**, che essendo esterna al waiter non comporta nessun vincolo sul numero dei suoi utilizzatori.

Difatti, in questa configurazione un aumento del numero di tavoli e/o di waiter andrebbe ad impattare in maniera minima sulle considerazioni qui fatte e sul prodotto che ne scaturirà.

Messaggi:

Request calculateTime: calculateTime(OK)
Reply time: time(TIME, TABLE)

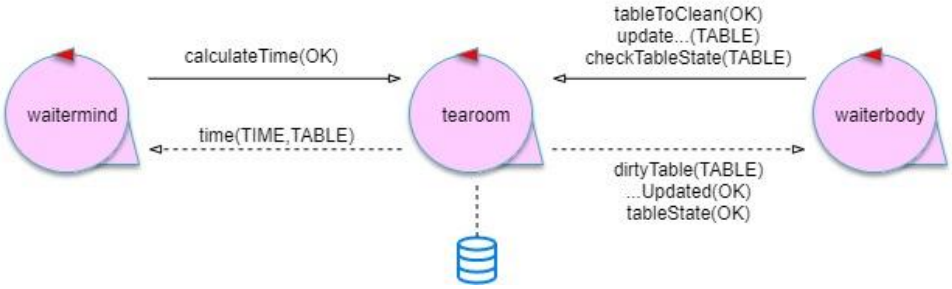
Request tableToClean: tableToClean(OK)
Reply dirtyTable: dirtyTable(TABLE)

Request updateDirty: updateDirty(TABLE)
Reply dirtyUpdated: dirtyUpdated(OK)

Request updateClean: updateClean(TABLE)
Reply cleanUpdated: cleanUpdated(OK)

Request checkTableState: checkTableState(TABLE)
Reply tableState: tableState(OK)

Tutti i messaggi non evidenziati riguardano lo scambio di messaggi fra **waiterbody** e **tearoom** riguardo lo stato dei tavoli.



Nello specifico tearoom ingloberà una base di conoscenza. Tale base può essere implementata in numero modi differenti:

- Uso di un database (ex. MySql)
- Uso di variabili interne all'attore
- Uso di appositi oggetti

- Uso di una base di conoscenza scritta in prolog

La nostra software house ha implementato un modo per poter comunicare con una base di conoscenza prolog direttamente dal qak actor, tale sistema fa uso del software tuProlog.

Di conseguenza, la strada da percorrere per tener conto dello stato dei tavoli è l'uso dell'attore tearoom che comunicherà attraverso delle query con la base di conoscenza, e aggiornerà lo stato dei tavoli tramite delle assert e retract.

Come rendere disponibile il current state

Sia che si decida di intraprendere l'approccio centralizzato sia che si opti per quello distribuito, nasce la problematica di come rendere disponibile il current state ad un osservatore esterno, per esempio il manager. Cioè come può un attore rendere disponibile il suo stato ad un'entità qualsiasi?

- **Polling:** l'entità aliena dovrà richiedere periodicamente, tramite messaggi (request/reply), il current state del sistema. È una soluzione poco elegante e precisa, in quanto comporta un grande traffico di messaggi e non garantisce di avere un current state aggiornato. Inoltre, non si è responsive a cambiamenti molto veloci del current state, che rischiano di essere persi. Questo approccio richiede che l'entità aliena conosca direttamente chi mantiene lo stato, creando un forte accoppiamento fra le entità.
- **Observer:** il current state del sistema è la risorsa observable e l'observer è l'entità aliena. Ogni cambiamento dell'observable è notificato all'observer, senza la necessità che l'observable sia a conoscenza dell'observer o degli observer.

Nella situazione di un sistema distribuito come quello che si sta trattando, la necessità è quella di creare un **observer remoto**. Nasce quindi la problematica di modellare un sottosistema che implementi i meccanismi di observer/observable remoti. Ovviamente un lavoro del genere richiederebbe una grande quantità di risorse ed un notevole lasso di tempo.

A questo proposito, il modello qak offre la possibilità di rendere **COAP-observable lo stato degli attori** e di aggiornare/notificare i cambiamenti di stato agli observer, tramite la funzionalità updateResource. In questo modo c'è la possibilità di far interagire un attore qak con entità aliene, senza dover costruire un'ulteriore componente software che si occupi di creare i meccanismi per observer/observable distribuiti.

Un'altra possibilità è di utilizzare un modello **pub/sub** come MQTT al fine di disaccoppiare ulteriormente gli observable dagli observer. Gli observable pubblicano su un topic noto i cambiamenti di stato ed essi verranno notificati a tutte le entità che si saranno iscritte a quel topic, senza dover sapere dove l'observable sia locato.

Problematica del waiter e della smartbell

Nello sprint 1 si è realizzato un waiter in grado di gestire un cliente alla volta, ma ciò non è applicabile al caso con più clienti.

La problematica deriva dal fatto che se un cliente desidera entrare all'interno della sala, ma il waiter sta eseguendo già un'azione (ex. pulire un tavolo), esso non riceverà risposta finché il waiter non termina il suo task.

La soluzione che si è valutata come più vantaggiosa è separare il waiter in due componenti:

- **Waitermind**: ha il compito di interagire con la smartbell per poter dare una risposta al cliente nel minor tempo possibile, esso consulta lo stato della stanza e risponde di conseguenza (se c'è un tavolo libero il tempo di attesa sarà zero, se ci sono tavoli sporchi avrà un certo valore e così via).
- **Waiterbody**: ha il compito di gestire la sale da the, andando a prendere le ordinazioni al tavolo, accompagnare i clienti e tutte le altre azioni richieste all'interno dei requisiti.

Per quanto riguarda la smartbell, le modifiche da fare non risultano eccessive, al momento dello sprint 1 è presente solo uno stato che si occupa di verificare la temperatura e comunicare al waiter tramite evento la presenza di un cliente.

Dai requisiti e dal confronto con il committente si evidenzia che la comunicazione migliore tramite waiter e smartbell sia tramite request/reply.

Inoltre, risulta necessario aumentare il numero di stati della smartbell per poter gestire il caso in cui la temperatura non vada bene, cioè che sia maggiore di 37.5 gradi.

WaiterMind nello specifico

```
Request waitTime: waitTime(ID)
Reply   waitTimeAnswer: waitTime(TIME)

Dispatch newClient: newClient(ID, TABLE)

Request calculateTime: calculateTime(OK)
Reply time: time(TIME, TABLE)
```

Waitermind comunica con la smartbell tramite request/reply al fine di fornire il tempo di attesa per la sala ed ha il compito di informare il **waiterbody** dell'ingresso di un nuovo cliente.

La **smartbell** richiede tramite *waitTime* il tempo di attesa per un cliente. Il **waitermind** inoltra la richiesta alla **tearoom** (a cui si rimanda più avanti riguardo *calculateTime* e *time*) tramite *calculateTime* e ottiene la risposta tramite *time.* A questo punto waitermind restituisce alla smartbell il tempo di attesa tramite la reply *waitTimeAnswer* ed informa il waiterbody di un nuovo cliente tramite il dispatch *newClient.*

Da notare che il waiterbody viene notificato della presenza di un nuovo cliente solo se il tempo di attesa è pari a zero: cioè sono presenti tutte le condizioni per far accomodare il cliente all'interno della sala.

WaiterBody nello specifico

```
Request moveToEntrance : moveToEntrance(ARG)
Request moveToTable : moveToTable(TABLE)
```

```
Request moveToBarman : moveToBarman(ARG)
Request moveToExit : moveToExit(ARG)
Request moveToHome : moveToHome(ARG)

Reply moveOk : moveOk(ARG)
```

La prima sezione del waiterbody si occupa di gestire il **movimento del robot** in base agli input ottenuti. Ad esempio, se il waiter è libero ed un cliente vuole entrare nel sistema, il waiterbody invia una richiesta *moveToEntrance* al **walker** per muovere il robot all'ingresso.

Per cui, **waiterbody** comunica tramite request con il **walker** i movimenti che desidera fare. Il walker, che contiene le coordinate effettive delle aree verso cui si desidera andare, una volta svolta l'operazione di movimento, invia una reply di fine movimento al waiterbody: *moveOk*.

```
Request tableToClean: tableToClean(OK)
Reply dirtyTable: dirtyTable(TABLE)

Dispatch startTimer: startTimer(OK)
Dispatch stopTime: stopTimer(OK)

Dispatch order: order(TEA, TABLE)
Request makeTea : makeTea( TEA, TABLE )
Reply teaReady : teaReady( TEA , TABLE)

Dispatch readyToPay: readyToPay(TABLE)
Dispatch moneyCollected: moneyCollected(AMOUNT)

Request readyToOrder: readyToOrder(TABLE)
Reply   atTable: atTable(TABLE)

Dispatch newClient: newClient(ID, TABLE)
```

La seconda sezione del waiterbody si occupa degli aspetti di **gestione della sala e dei clienti**. Tramite la request *tableToClean*, richiede alla **tearoom** la presenza di tavoli da pulire e viene informato sul numero di tavolo tramite reply *dirtyTable*.

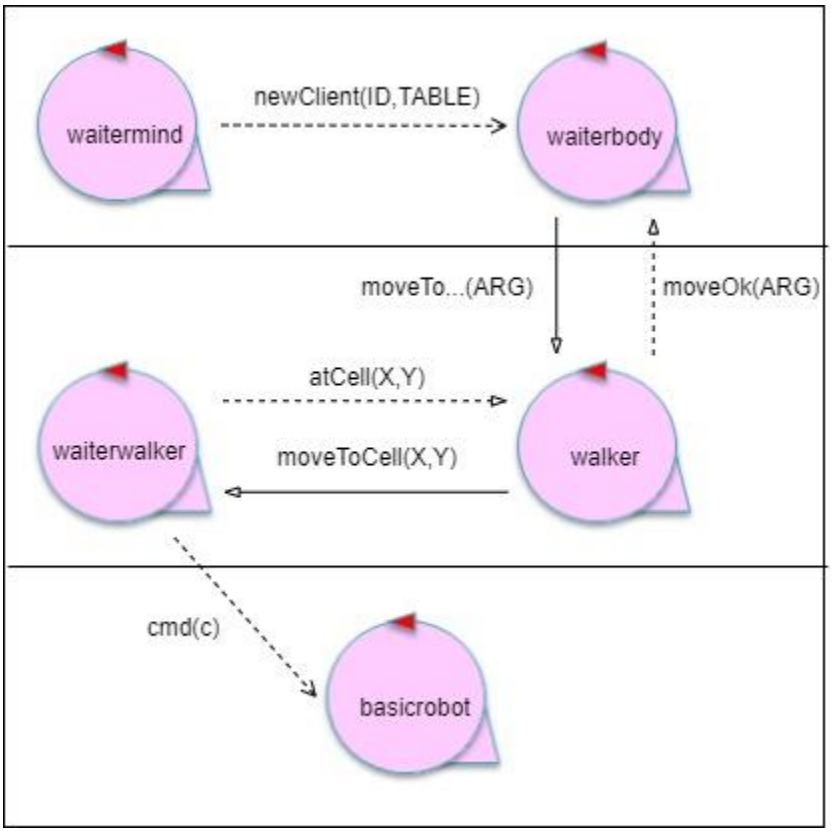
Gestisce la fase di ordine e di pagamento dei clienti ed il loro tempo massimo all'interno del sistema. Riguardo il tempo massimo all'interno del sistema, si rimanda al paragrafo più in avanti.

```
Request checkTableState: checkTableState(TABLE)
Reply tableState: tableState(OK)

Request tableToClean: tableToClean(OK)
Reply dirtyTable: dirtyTable(TABLE)

Request updateDirty: updateDirty(TABLE)
Reply dirtyUpdated: dirtyUpdated(OK)

Request updateClean: updateClean(TABLE)
```

Reply cleanUpdated: cleanUpdated(OK) Request checkTableState: checkTableState(TABLE) Reply tableState: tableState(OK)

L'ultima "sezione" logica è quella adibita allo scambio di messaggi con **tearoom** riguardo lo stato dei tavoli.

Si riporta un grafico riassuntivo delle varie parti del **waiter** e della comunicazione fra di esse (quindi si considerino solo i messaggi scambiati fra esse).



Actors e macrostati

Di seguito vengono presentati i vari attori illustrandone gli **stati.**

Waiter

newClient	In questo stato il waiter si dedica alla gestione di un nuovo client. Permane in tale stato dall'arrivo della notifica di un nuovo cliente fino al momento in cui il cliente viene fatto accomodare al tavolo libero.
orderManagement	Questo stato comprende tutte le azioni necessarie a far sì che un client possa ordinare e che il barman possa prendere in carico l'ordine.

serving	Quando il waiter si trova nello stato “serving” la sua mansione principale è recuperare il tea dal barman e servirlo al cliente che lo ha ordinato.
endClient	È lo stato in cui il client prima paga il waiter e successivamente viene accompagnato all’uscita.
cleaning	In questo caso il waiter si occupa di pulire i tavoli che non sono stati ancora igienizzati.
waiting	Questo è lo stato che indica la disponibilità del waiter a gestire le richieste che possono provenire dai clienti, dal barman o dalla smartbell.
atHome	Il waiter, dopo un periodo di inattività, si dirige alla home ed aspetta nuove richieste.

Smartbell

waiting	La smartbell è pronta a ricevere richieste di ingresso da parte dei clienti
checkingClient	*Controlla la temperatura di un cliente ed agisce di conseguenza in base al valore misurato*

Barman

waiting	Tale stato indica la disponibilità del barman ad accettare le richieste da parte del waiter.
makingTea	In questo caso il barman sta preparando un tea.

Microstati nel waiterbody

È possibile suddividere i macrostati individuati in microstati più precisi.

Stato	**Microstati**
newClient	• GoToEntrance • ConvoyToTable
orderManagement	• GoToTakeOrder • TakeOrder • sendOrder
serving	• TakeTea • Serve
endClient	• GoToCollect • Collect • goToExit

cleaning	• GoToClean • clean
waiting	• -
atHome	• -

Problematica del tempo di attesa minimo

L'obiettivo principale del waiter è che i **clienti attendando il minor tempo possibile prima di accomodarsi** nella tearoom, come espresso nei requisiti. Da ciò nasce l'esigenza di poter **interrompere**, se possibile, un determinato task (azione) che il waiter sta svolgendo se sono soddisfatte le condizioni per fa accomodare un client.

Ad esempio, si consideri la situazione nella quale il waiter stia pulendo il tavolo1 ed il tavolo2 sia pulito e libero. A questo punto si ipotizzi che un client notifichi la volontà di entrare nel sistema e che lo smartbell lo accetti. In modo da poter ridurre al minimo la sua attesa, il waiter potrebbe interrompere il lavoro di pulizia e far accomodare il client, per poi proseguire il lavoro successivamente.

Allo stesso modo, se in una situazione simile un client A desideri pagare ed uno B accomodarsi nel sistema, il waiter dovrà preferire far accomodare nel sistema il cliente A in arrivo e poi servire il client B, tenendo in considerazione sempre il tempo di esecuzione dell'azione che si sta svolgendo.

Da questo aspetto del problema, si deduce che i task abbiamo delle **priorità** di esecuzione e possano essere **interrompibili** o meno.

Si analizzano i task elencati nei requisiti.

Task	Interrompibile	Opportuno?	Priorità	Spiegazione
accept	no	-	alta	-
inform	no	-	alta	-
reach	no	-	alta	-
take	si	no	media	La durata dell'azione è contenuta.
serve	si	no	media	La durata dell'azione è contenuta.
collect	si	no	media	La durata dell'azione è contenuta.
convoy	no	-	media	Non è possibile lasciare il client in una posizione indefinita della tearoom per questioni igieniche.
clean	si	si	bassa/media	Si può interrompere il lavoro in uno stato definito (si veda il paragrafo precedente riguardo lo stato di un

				tavolo) e continuarlo in seguito.
rest	si	si	bassa	Il waiter è nello stato atHome, pronto a ricevere richieste.

Calcolo del tempo di attesa di un cliente

La problematica del tempo di attesa non è centrale, in quanto deve essere un'idea del tempo che il cliente deve aspettare e non è un dato vincolante. Per il calcolo del tempo di attesa servono informazioni riguardanti lo stato dei tavoli e del waiter.

Per cui il tempo di attesa può essere fornito dall'actor **tearoom** che contiene tutte le informazioni relative alla sala. Ricapitolando lo scambio di messaggi, il **waitermind** chiederà tramite request *calculateTime*, alla tearoom il tempo di attesa, ottenendo la risposta tramite *time* (si vedano i messaggi evidenziati in giallo nel paragrafo precedente).

Problematica del maxStayTime al tavolo

Per tenere traccia del maxStayTime è possibile seguire in linea di massima due approcci:

- **Centralizzato:** introdurre un'entità ulteriore per gestire tutti i "timer" relativi ai singoli tavoli-clienti.
- **Distribuito:** introdurre un timer per ogni tavolo.

L'approccio distribuito, che comporta l'introduzione di due attori **timer1** e **timer2** si dimostra più semplice nel caso attuale in cui sono presenti solo due tavoli. L'approccio centralizzato, che introduce un sottosistema per la gestione dei timer, richiede più lavoro di coordinazione, ma sicuramente si presta più facilmente all'espansione della sala, cioè all'introduzione di altri tavoli.

Data la natura del problema attuale, risulta più semplice limitarsi all'introduzione di due timer.

```
Dispatch startTimer: startTimer(OK)
Dispatch stopTime: stopTimer(OK)

Dispatch readyToPay: readyToPay(TABLE)
```

Il **waiterbody** avvia i timer tramite *startTimer* e li stoppa se necessario tramite *stopTime*, cioè se il client notifica la volontà di pagare prima di maxStayTime. Se il timer scade, il waiterbody viene notificato tramite *readyToPay(TABLE)* dal timer e viene forzato a richiedere il pagamento al cliente al tavolo n°TABLE.

Modello

Disponibile presso: https://github.com/virtualms/IssProject/tree/master/sprint2/modelProblemAnalysis

---

TEST PLAN

Un test significativo riguarda il **tempo massimo in cui il cliente può sostare nella stanza da tea** e il relativo cambiamento di stato dei tavoli.

Per fare ciò si controlla preventivamente che i tavoli siano entrambi nello stato clean. Successivamente si verifica ogni cambiamento di stato relativo all'azione di ingresso di 2 clienti, di consumazione e di igienizzazione dei tavoli.

Nello specifico, sarà simulato che un cliente chieda di pagare, mentre per il secondo sarà verificato che il waiter gli chieda di pagare ed uscire dalla sala.

Tutto ciò, sarà testato andando, come detto, a controllare gli stati relativi ai tavoli e lo stato del waiter.

TEST PLANS

I test plan rimarcano quanto introdotto nella fase di analisi dei requisiti. Si controlli quindi che il sistema, ad un certo stimolo di un client, risponda nella maniera aspettata.

Consultabili presso: https://github.com/virtualms/IssProject/tree/master/sprint2/test.

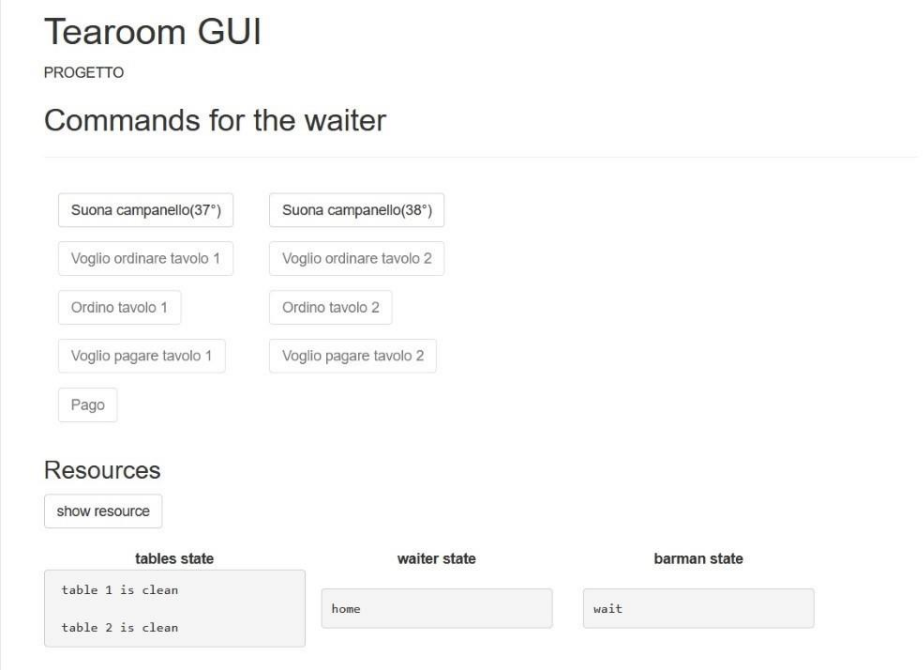
PROGETTO

Grazie all'utilizzo dei qak in fase di analisi, è possibile ottenere immediatamente un prototipo eseguibile in Kotlin del sistema descritto durante la fase di analisi del problema.

Per il modello consultare: https://github.com/virtualms/IssProject/blob/master/sprint2/system.qak

Manager

Il current state del sistema è visualizzabile dal manager attraverso un'apposita **web app**, sviluppata utilizzando **Spring Boot.** La web app è fornita inoltre di un comparto di simulazione di clienti, in modo da rendere possibile al committente un test interattivo del sistema.

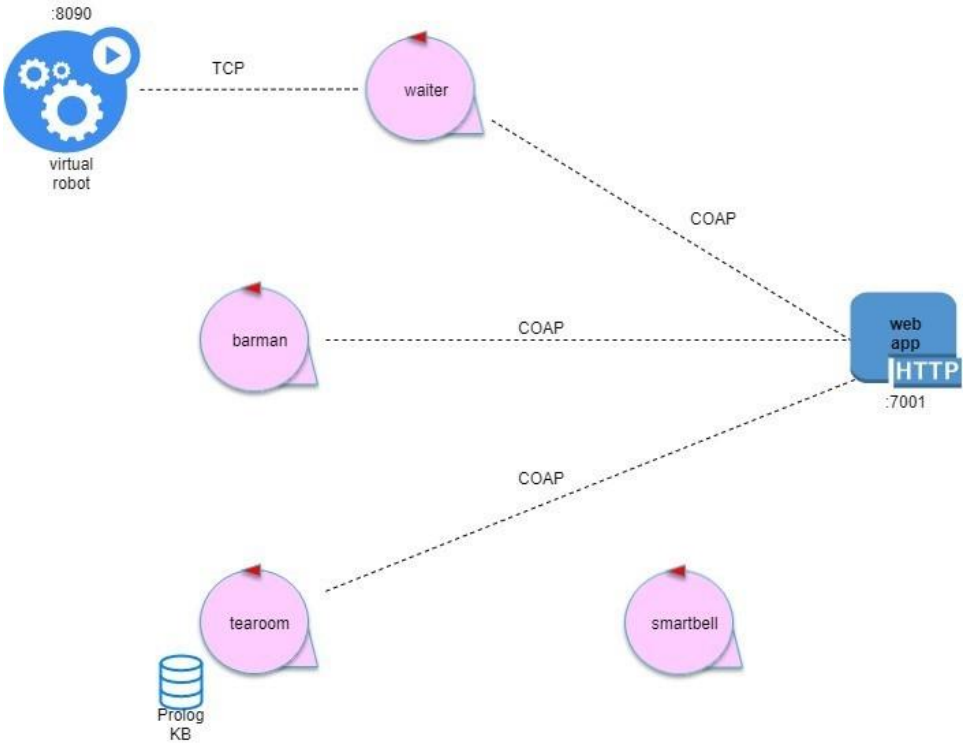


Base di conoscenza

Si è scelto di implementare una base di conoscenza in **prolog**, considerando la vicinanza del modello qak ad esso. Per cui tearoom opererà sullo stato dei tavoli attraverso delle *assert* e *retract e* otterrà le informazioni necessarie attraverso opportune query prolog.

Composizione del sistema

Una visione sicuramente *informale* del sistema, ma che da immediatamente un'idea della sua composizione a grandi linee è la seguente. Per chiarezza del grafico sono stati *omessi gli attori che compongono waiter*, precedentemente discussi, e le comunicazioni fra i vari attori. Inoltre, *non vengono rese esplicite tutte le comunicazioni* fra webapp, virtual robot e tearoom system(barman, waiter, smartbell, tearoom), per chiarezza espositiva.



Deployment

In questo sprint tearoom system (e quindi tutti gli attori confinati nel sistema) e virtual robot risiedono sulla macchina A. La web app può risiedere su una macchina B ed interagire da remoto con il tearoom system tramite COAP e web socket per simulare un client ed ottenere le informazioni relative allo stato del sistema.