LABORATORIO DI SISTEMI SOFTWARE

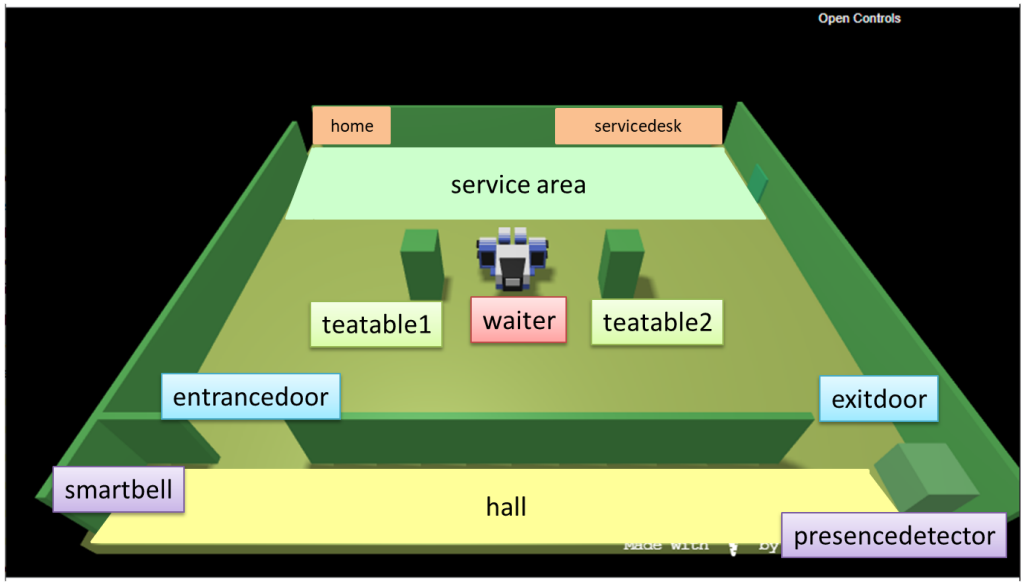
Introduction

The ***manager*** of a ***tearoom*** intends to regulate the access to the service by means of a ddr robot (***waiter***).

The ***tearoom*** is a rectangular room that includes:

- an ***entrancedoor*** to enter in the room and an ***exitdoor*** to exit form it;
- a number N (N=2) of ***teatable***;
- a ***serviceare*** including a ***servicedesk*** at which works a ***barman***;
- a ***hall*** equipped with a ***presencedetector***, i.e. a device (e.g. a sonar) that can detect the presence of a person (or some other entity) in it.

The ***waiter*** can freely move along the borders of the tearoom, since there are no obstacles there.



User stories

As a ***client***,:

- I intend to ***notify*** my interest in ***entering*** in a ***safe tearoom***, ***sitting*** at a free teatable, ***ordering*** some tea, ***consuming*** it (within a limited amount of time ***maxstaytime***) ***paying*** the service with my credit card and finally ***leaving*** the room.
- For ***safe tearoom***, I intend a tearoom with ***clean*** tea-tables posed at a proper distance; the room is populated by human clients whose body temperature is less than 37.5 degrees.
- I can submit my notification of interest by hitting the ***smartbell*** located near the ***entrancedoor*** that will automatically measure my body temperature and send a request message to the ***waiter***, by giving to me an unique ***clientidentifier***.
- I my body temperature is ok, but my request cannot be immediately satisfied (since the room is full), I will be ***informed*** by the ***waiter*** aabout the maximum waiting time.

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.

Requirements

The ***waiter*** should perform the following tasks:

- ***accept*** the request of a client to enter in the tearoom if there is at least one teatable in the state ***tablelean***, i.e. the table is free and has been properly cleaned;
- ***inform*** the client about the maximum waiting time if there is no ***tableclean***;
- ***reach*** the ***entrance door*** and ***convoy*** the accepted client to the selected teatable;
- ***take*** the order of the client and transmit it (using a wifi-device) to the ***barman***;
- ***serve*** the client when the ***barman*** says that the requested drink is ready;
- ***collect*** the payment from the client when he/she has finished to consume or when the ***maxstaytime*** is expired;
- ***convoy*** the client to the ***exitdoor***;
- ***clean*** the tea-table just freed by the client;
- ***rest*** at my ***home*** when there is nothing to do.

Since the room could contain N clients at the time, the ***waiter*** should reduce as much as possible the waiting time of the requests coming from each client.

Optional: one client in the hall

The ***waiter*** must open the ***exitdoor*** only when the hall is free, i.e. it must not open that door if the hall is already engaged by a client waiting to enter at the ***entrancedoor***.

Non functional requirements

1. The ideal work team is composed of 3 persons. Teams of 1 or 2 persons (**NOT** 4 or more) are also allowed.
2. The team must present a workplan as the result of the requirement/problem analysis, including some significant **TestPlan**.
3. The team must present the sequence of SPRINT performed, with appropriate motivations.
4. The team must present (in synthetic, schematic way) the specific activity of each team-component.

Requirement analysis

Dai requisiti si evince che il ***waiter*** deve effettuare una serie di task diversi. Ogni tavolo può essere in diversi stati, per esempio sporco o pulito. Per una fase iniziale è opportuno modellarlo come oggetto, in quanto facilita la gestione dei test e non impone vincoli implementativi o progettuali. Infatti, potrebbe essere comunque sviluppato come una qualunque altra entità, nel caso in cui venisse successivamente creato un layer superiore.
Dopo aver interpellato il committente, la gestione del "***maxstaytime***" è la seguente: dopo aver portato il cliente al tavolo, faccio partire il tempo. Da quel momento in avanti, la misurazione viene fatta solamente quando è responsabilità del cliente il fatto di "perdere" tempo. Per esempio, se il cliente non fa mai un ordine, la colpa è del cliente stesso, non del ***waiter*** o del ***barman***. Dopo aver fatto l'ordine, il tempo non viene fatto scorrere in quanto quello che passa non è più "colpa" del cliente ma del ***barman***. Quindi il tempo scorre solo quando è responsabilità del cliente.
Con il committente è stato chiarito anche il concetto di "current state" della tearoom. Essa identifica lo stato di tutte le entità e componenti all'interno del sistema, per esempio la posizione e il task attuale del ***waiter***, lo stato dei tavoli o l'occupazione del ***barman***.
I messaggi che circolano nel sistema sono i seguenti:

- Request/Response per la richiesta di un cliente per entrare e la relativa risposta, positiva o negativa, data dal ***waiter***

- Dispatch per i messaggi di ordine pronto, trasmissione ordine al ***barman*** e per la richiesta di pagamento del cliente.

Problem analysis

Dall'analisi dei requisiti nasce il problema della gestione delle entità e dei loro stati. Il ***waiter***, per esempio, è un'entità con comportamento autonomo: è quindi opportuno modellarlo come un attore. Diversa considerazione deve essere fatta per i tavoli. Essi hanno uno stato, come espresso dai requisiti, ma è necessario esprimerlo formalmente. Modelliamo il tavolo come un oggetto, così che i tavoli siano istanze di tale oggetto, che contiene una proprietà che fa da "stato" del tavolo, utilizzando una stringa, per rendere eterogeneo il sistema. Nel caso in cui, in sprint successivi, venisse fuori una problematica che induce di modellare il tavolo come attore, basterebbe inglobare il modello a oggetti in un attore di livello superiore, così da interoperare utilizzando uno scambio di messaggi.

Un'altra problematica che sorge è come gestire lo stato delle risorse: distribuito o concentrato. Nel distribuito c'è un ente che è l'unico che conosce il suo stato, nel concentrato una risorsa conosce lo stato delle varie entità che compongono il dominio. Partendo dal presupposto che la modellazione delle risorse in modo distribuito è da evitare perchè è difficile mantenere coerenza e consistenza, in questa problematica è opportuno utilizzare un approccio misto in cui rappresentiamo lo stato del mondo con una risorsa "world" in cui il sistema è fatto da due entità fondamentali, ***waiter*** e "world" intorno a lui. La conseguenza diretta di questa analisi riguarda come gestire queste due entità, ovvero gestirle indipendentemente oppure inserire il "world" all'interno del ***waiter***. Nel caso in questione, i requisiti ci indicano una via: ogni tipologia di richiesta/messaggio viene fatta verso il ***waiter***, il quale deve avere la possibilità di accedere velocemente allo stato del "world". Infatti, se facessi due entità separate, il ***waiter*** dovrebbe continuamente interagire con l'entità esterna, comportando un onere computazionale e un traffico non indifferente di comunicazione. Quindi la problematica dovrebbe essere modellata introducendo un'unica entità ***waiter*** il quale contiene al suo interno lo stato "world".

Il ***waiter*** è un'entità che nel momento in cui fa un task, potrebbe dover sentire dei messaggi che gli arrivano dal mondo esterno. Se zoommiamo dentro al ***waiter*** è opportuno che ci siano entità separate e specifiche, magari una serie di attori. In una fase iniziale è sufficiente pensare al ***waiter*** come un insieme di due attori, uno attento al modno esterno e uno attento alle operazioni da fare. Ovvero il ***waiter*** deve essere proattivo, ovvero deve poter fare cose per contro proprio, ma anche reattivo, cioè riuscire a catturare tutte le informazioni del mondo esterno, ovvero deve essere consapevole di tali informazioni. Per esempio se abbiamo una coda di messaggi in ingresso mentre il ***waiter*** sta pulendo il tavolo, possiamo dire che siamo sicuri che se l'informazione gli arriva nella coda, il ***waiter*** può finire di effettuare il task che sta facendo sicuro che non siano perdute tali informazioni arrivate.

Una problematica cruciale che viene direttamente dall'analisi appena effettuata se sono nella parte proattiva è se può essere opportuno che il ***waiter*** interrompa il task che sta facendo per farne un altro più prioritario. E' necessario prendere uno per uno tutti i task, analizzarli e capire se e quali sono i task interrompibili.

Task	Tempo nel caso peggiore(sec)	Interrompibile(sì/no)	Descrizione	Caso
Accept	5	no	è un task leggero e veloce, si tratta di un'accettazione di richiesta, quindi la sua durata potrebbe essere messa tra le durate "minime" di tutti i task presenti nel sistema, durata breve. Per finire questo task non è necessario molto tempo, quindi non ha senso interromperlo.	-
Inform	5	no	task leggero e veloce, durata breve. Per la	-

			stessa considerazione non interrompibile.	
reach and Convoy	80	sì	task in media più duraturi di tutti quelli presenti nel sistema. Stimiamo con il committente che possa volerci nel caso peggiore circa 80 secondi per effettuare il reach verso la entrance door (da servicedesk caso peggiore). **Nasce una problematica**: nel caso in cui il ***waiter*** stia accompagnando un cliente ad un tavolo, non è stato ancora aggiornato lo stato del tavolo stesso. Questo vuol dire che se nel mentre arriva una richiesta di un altro cliente per entrare, è opportuno non interrompere il task attuale, per non entrare in una inconsistenza di stati (se accettassimo quel cliente, potrebb instaurarsi una catena infinita di accettazioni di clienti, non aggiornando mai lo stato del tavolo). Reach and convoy possono quindi essere interrompibili solo per richieste che NON siano nuovi clienti in arrivo.	ricezione richieste: serve, collect payment, take o convoy
Take	20	no	Il ***waiter*** va a prendere l'ordine del cliente e informa il ***barman***. Con il committente è stato deciso che non è un task interrompibile.	-
Serve	30	no	Il ***waiter*** si reca a prendere l'ordine pronto e lo porta al cliente. Non è interrompibile.	-
Collect payment	50	no	durata medio/breve, sui 50 secondi. Con il committente, per questioni di marketing, è stato deciso che questo task non sia interrompibile.	-
Convoy	50	no	Se il ***waiter*** deve fare Convoy verso exitdoor il caso peggiore è dal tavolino più lontano, stimiamo con il committente che questo possa essere minore come tempo, circa 50 secondi. Con il committente è stato deciso che convoy non è interrompibile	-
Clean	60	sì	task da considerare con il committente per la durata, circa 60 secondi, quindi interrompibile. Con il committente è necessario stabilire se ci sia un limite entro cui interrompere questo task, per esempio un limite a livello di "stato" in cui si trova, oppure, se è opportuno che il task venga interrotto in ogni momento	ricezione di qualunque richiesta
Rest	variabile	sì	Il ***waiter*** rimane in attesa di richieste nella sua home, quindi è il task interrompibile per eccellenza.	ricezione di qualunque richiesta

Un esempio potrebbe essere che mentre il ***waiter*** fa il task ***clean*** per un cliente è pronto del cibo o il cliente è pronto a pagare: potrebbe essere il caso di interrompere e proseguire dopo aver completato l'altro task (prioritario). Ma se ho effettuato da tanto tempo il ***clean*** e mi manca poco a finire, potrebbe essere necessario concordare un limite entro il quale un task è interrompibile o no. Dobbiamo concordare con il committente se per lui può andare bene oppure che il task venga interrotto comunque.

E' opportuno modellare lo stato del tavolo all'interno della "mente" del ***waiter***, ovvero una componente che memorizza un determinato stato e riesce a far orientare il ***waiter***, soprattutto per task interrotti, per poterli ripristinare da dove era arrivato a farli. In questo modo, inoltre, semplifichiamo il modello in quanto non introduciamo altre entità riguardanti il tavolo. Per modellarlo, è opportuno utilizzare una stringa che rappresenta lo stato, il quale può essere:

- "dirty": tavolo sporco, è appena andato via il cliente.
- "undirty": tavolo pulito dai residui del cliente precedente, quali stoviglie e tovaglia.
- "sanitized": tavolo pulito con prodotti disinfettanti e tavolo sanitizzato.
- "cleaned": tavolo rimesso in sesto con tovaglia e stoviglie nuove, pronto per accettare un nuovo cliente.

Quindi abbiamo una rappresentazione esplicita di stato, che passa da "dirty" a "cleaned" passando da delle fasi intermedie. Questa esigenza è dovuta dal fatto che dobbiamo sapere quale sia lo stato del tavolo, perchè se il ***waiter*** interrompe questo task, deve riprendere dallo stato in cui era arrivato e non ripartire da capo. Il ***waiter*** comincia a fare "***clean***" e, mentre lo fa, lo stato del tavolo cambia, passando, se non succede niente, da tutti i quattro stati. Queste problematiche affrontate aiutano a modellare il sistema in modo eterogeneo e indipendente dall'implementazione futura, in quanto ogni tecnologia riesce a maneggiare una stringa.

Test plans

Un test significativo per questo sistema è quello di simulare l'esecuzione di un task interrompibile, far avvenire un evento/messaggio che induca il task ad interrompersi e valutare se il funzionamento sia il seguente: il ***waiter*** smette, salva lo stato in cui era arrivato, esegue l'altro task e, dopo averlo concluso, riprende da dove ero arrivato nel primo task. Un altro test significativo è quello di simulare più richieste simultanee mentre il ***waiter*** sta effettuando un task interrompibile. Nel caso in cui ho un tavolo occupato, il ***waiter*** sta pulendo un tavolo e arriva una richiesta di un cliente, il ***waiter*** deve negare la richiesta di ingresso, perchè non ho più tavoli, e informarlo del tempo di attesa rimanente. Ma se il ***barman*** ha già preparato la consumazione mentre sto pulendo, devo interrompere il task e tornare lì.

Il test plan consiste nel simulare l'arrivo di un altro messaggio o richiesta, per esempio il ***barman*** informa che l'ordine richiesto è pronto per la consumazione del cliente. A quel punto il ***waiter*** non fa più il task che stava facendo, va a servire l'ordine e riprende successivamente il task precedente. Per poter modellare questi test, è necessario aver modellato il funzionamento del ***waiter***, quindi il ***waiter*** deve essere consapevole di fare quei determinati task.



Paolo Verdini, Angelo Farina, Riccardo Turra

paolo.verdini@studio.unibo.it, angelo.farina@studio.unibo.it, riccardo.turra@studio.unibo.it