modalità col quale client-server interagiscono tra loro

Definiamo un protocollo di comunicazione a messaggi per comunicare tra client e server,

in modo tale che capiamo quali informazioni ci arrivano dal client, come gestire gli eventi realtime (scelte in tempo reale), e **come devono comportarsi i metodi del controller.**

L'applicativo CLIENT è in grado di inviare messaggi al server che possono essere di tipi stabiliti

Ad esempio:

messaggi di servizio:

MSG_ERROR

MSG_OK_GENERIC

messaggi per la creazione lobby (all'avvio del programma)

"CREATE localhost 1337 4"	->	MSG_CREATE_LOBBY
"JOIN localhost 1337"	->	MSG_JOIN_LOBBY

messaggi di gioco:

- QUANDO È IL TURNO di gioco di un giocatore, l'app Client riceve dalla interfaccia di input (stdin) dei comandi.
  Con l'interfaccia grafica immaginiamo che ci siano dei bottoni, e son sempre quelli.
- **A ogni comando** corrisponde un messaggio che verrà inviato al server.
  Ogni messaggio avrà associate anche un pacchetto di informazioni, che il server userà per sapere cosa fare.

Durante il turno possono essere eseguite diverse azioni, e per far ciò:

>> il SERVER sta in ascolto del messaggio (messaggio -> comando -> azione)

>> il CLIENT prende il comando da input e ci costruisce sopra il suo messaggio:

1. azione di prendere risorse dal market -> G_MARKET_ACTION

(come funziona l'azione di PRENDERE RISORSE AL MERCATO)

>> il client comunica al server la richiesta, specificando nel messaggio riga o colonna (e il numero di riga o colonna) (esempio: messaggio di tipo G_MARKET_ACTION, e castando il messaggio secondo G_MARKET_ACTION possiamo accedere al campo “numero”, o “isColonna”).

>> il server recepisce il messaggio, CAPISCE l'azione corretta da parte del giocatore (ossia che vuole prendere risorse dal mercato), e invoca il metodo corretto del controller con i parametri desiderati.

>> il client resta in ascolto della risposta (che può essere):

>> messaggio dal server: "cosa vuoi fare? 1: scartare, 2: mettere nel deposito [bloccata], 3: scambiare righe del deposito....

(il server calcola in anticipo l'insieme delle azioni che il client può fare, e formatta il messaggio di richiesta in seguito)

(il server fa questa richiesta per OGNI biglia, fino a che le biglie non sono state scartate oppure inserite nel deposito)

(la biglia rossa è gestita a parte)

>> messaggio dal server: errore! (o qualcosa di simile, azioni non consentite)

>> messaggio dal server: FINE

>> dopo il messaggio FINE, il client attende un nuovo comando dall'utente

2. azione di finire il turno: -> G_END_TURN

>> il client invia il messaggio, riceve l'OK dal server, e ritorna in ascolto perenne.

(si mette in ascolto per ricevere il messaggio di "è il tuo turno!" oppure.. aggiornamenti della sua view (del client))

3. azione di attivare le leaderCard -> G_ACTIVATE_LEADER_CARD

>> il client comunica al server la richiesta, specificando quale carta vuole attivare

>> il server verifica che si possa attivare, e in tal caso la attiva inviando una risposta positiva.

4. azione di spostare le risorse nel deposito -> G_CHANGE_DEPOT_CONFIG

>> il client comunica tutta la 'nuova' configurazione del deposito

>> il server riceve la configurazione, la valida e in caso aggiorna il model del player, inviando risposta positiva.

5. azione di comprare UNA carta sviluppo -> G_BUY_CARD

>> il client comunica L'INTENZIONE di comprare una carta sviluppo

>> il server, che **sa** quali sono le carte rimaste **e** le risorsè disponibili del giocatore,

calcola quali il giocatore può comprare e le invia al client in un messaggio che contiene..

una lista, del tipo: [carta 1 e la puoi mettere nello stack 2]

[carta 2 e la puoi mettere nello stack 3]

>> il client SCEGLIE uno di questi record.

>> il server, ricevuto il record, sa quale carta vuole comprare e dove la vuole mettere.

>> il server PROCEDE a rimuovere le risorse DAPPRIMA dal deposito del giocatore (e poi dal forziere), e inserisce la carta nel suo DevelopmentCardSlot, e invia una risposta positiva al client.

6. azione di attivare le produzioni -> G_ACTIVATE_PRODUCTION

>> il client invia un messaggio al server che contiene 'quali carte il giocatore vuole attivare'

>> il server, recupera i riferimenti alle carte specificate, e crea dei CALDERONI:

un CALDERONE per le risorse di input (per verificare se il client attualmente le ha.)

un CALDERONE per le risorse di output (ottenute applicando tutti i vari 'power' delle carte di produzione/leader)

>> se il giocatore dispone di quelle risorse, attua la produzione, inviando risposta positiva.

(situazioni particolari come quelle di scegliere una risorsa a scelta vengono risolte chiedendo al client, il quale è in ascolto di risposta)

(anche in questo caso le risorse che stanno nel CALDERONE di input vengono consumate prima dal deposito e poi dal forziere)

Ci saranno dei “manager” nella parte di controller che servono a gestire interi aspetti del programma. In particolare:

il FaithTrackManager per gestire il tracciato papale. Esso fa tutta una serie di controlli e manipola il model. Viene chiamato dalle “azioni” sopra descritte.

Il GameManager, che gestisce i turni, i giocatori correnti (e probabilmente anche la parte di guidare Lorenzo El Magnifique in solitaria)

Il gestore delle connessioni, e il layer che fa il parsing dei messaggi ricevuti e chiama le azioni corrispondenti corrette (più avanti)

Il controller vero e proprio, che contiene le nostre azioni.

Il client di per sé un THIN client, e dalla sua parte c’è una logica di sanitizzazione dell’input e di correttezza delle informazioni inserite.

Domanda: se volessimo stendere il codice per scrivere queste azioni, dovremmo ricevere le informazioni dalla rete. Tuttavia questa parte è difficile da fare per ora. Dovremmo FORNIRE NOI le informazioni al controller, chiamare i suoi metodi, e usare input/output da console, per poi sostituirli e mettere la parte di rete?

AZIONI VERE E PROPRIE (codice, funzionamento del codice), a livello di codice:

1. prendere_risorse(boolean Row, int num)

- da input, Row può essere vero (e in tal caso è una row) oppure falso (e in tal caso è una column). Num può variare da 1 a 3 se è riga o da 1 a 4 se è column

Creiamo una lista temporanea di MarketMarble, chiamata temp.

Capiamo se è una riga o una colonna, e in caso chiamiamo il metodo Market.getRow(num) (o column)

Creiamo anche una lista di Risorse, chiamata R.

PER OGNI BIGLIA IN TEMP, chiamiamo il suo metodo .doAction(R).

Se è una biglia normale, aggiunge una risorsa a R. Se è una biglia bianca, aggiunge una risorsa EXTRA a R. Se invece è una biglia rossa, solleva una eccezione che viene gestita invocando un metodo apposito di FaithTrackManager.

A questo punto in R abbiamo tutte le risorse da aggiungere.

PER OGNI BIGLIA chiediamo all'utente cosa vuole farne: (ad esempio). Calcoliamo COSA può fare (guardando i depositi, e i depositi extra, e le leaderCards) e:

- se è una risorsa EXTRA, dipende dalla leaderCards (lo scambio potrebbe essere fatto subito)
- se è una risorsa NORMALE, può metterla nel deposito, scartarla, metterla nel deposito extra
- l'utente può scambiare le righe del deposito, oppure di skippare una risorsa per farla dopo.

Riceviamo la risposta e agiamo di conseguenza, fino ad esaurimento delle risorse in R.

2. end_turn(Player p?)

il metodo invoca il manager dei turni, che gestisce la situazione. (giocatore corrente, numero di turno, controlli se la partita è finita, etc.)

3. activate_leader_card(int quale)

il controller capisce quale carta deve attivare, recupera i requisiti (di carta o di risorse) e fa il check. Se va a buon fine, flagga la LeaderCard contenuta nel model del player.

4. change_config(Resource shelf1, Resource[] shelf2, Resource[] shelf3, int X1, int X2)

- X1 e X2 rappresentano il numero delle risorse nei depositi extra (potrebbe essere ad esempio -1 se non ha un deposito, 0 se 0 risorse, e non può valere più di due)

Dobbiamo capire se, data la vecchia configurazione del giocatore (che andiamo a recuperare), e la nuova configurazione (che ci viene passata), le due hanno.. lo stesso numero di risorse, la disposizione nel depot è valida……..

5. Buy_card()

Generiamo una lista di DevelopmentCard, chiamata Temp.

Il controller **sa** quali carte sono 'visibili' nel DevelopmentCardDeck e le aggiunge alla Lista Temp.

Chiamando il metodo getTotalResource() del player, andiamo a ricavare una mappa di tutte le risorse che il giocatore possiede.

A questa lista, andiamo a togliere quelle che il player non si può permettere, confrontando il risultato di getTotalResource() con il costo della carta.

Filtriamo la lista, tenendo solo le carte che rendono vero il metodo validateNewCard() del developmentSlot del giocatore.

Quindi, creiamo una struttura a record (???) del tipo: [carta, posizione slot x]

Facciamo visualizzare questa struttura a record al client.

Il client quindi fa una scelta tra i record che visualizza, e il controller fa:

Per ogni risorsa (contenuta nel costo della carta), prova a consumare dal Depot del player. Se non c'è, prova a consumarla dall'ExtraDepot. Se non c'è, la consuma dallo Strongbox.

Chiedere?

Aggiunge la carta al DevelopmentSlot del player usando il metodo addCard.

6. Activate_productions(bool[3] choices, obj[3] leader/base)

- Obj : [bool enable, Resource[] input, Resource output]
- L'array choices rappresenta le carte sviluppo del developmentslot.
- L'array obj (da rivedere) ha un campo enabled (che fa la stessa funzione di choices) e le risorse scelte dal client.

Andiamo a creare un oggetto CALDERONE (una HashMap) nel quale mettiamo tutte le risorse chieste in input alle produzioni scelte.

Per choices: se uno dei valori è true, andiamo a recuperare la carta nel DevelopmentSlot nella posizione corrispondente (e quindi il suo costo)

Per le altre: i costi sono specificati negli oggetti.

Confrontiamo questo CALDERONE con il risultato del metodo getTotalResources() del giocatore.

Se vediamo che l'operazione è lecita, allora creiamo un secondo CALDERONE con tutte le risorse di output delle produzioni scelte (in modo simile alla creazione del primo CALDERONE), e le aggiungiamo tutte quante al forziere.

Nel caso di una produzione che dia una risorsa fede, chiamiamo il FaithTrackManager.