**L’Interfaccia Utente**

**1. Preambolo**

Nell’intento di mostrare al cliente un software funzionante, e al contempo di creare una struttura che renda facile un lavoro futuro di espansione e di evoluzione del software, i nostri sforzi si sono concentrati in queste prime iterazioni nello sviluppo di un modello di interfaccia quanto più flessibile ed incrementale, al compromesso di implementare l’effettiva interfaccia – seppur momentaneamente – in sola forma testuale.

L’interfaccia è uguale per ciascun Utente, seppur specializzandosi in base al tipo di Utente che la utilizza (Cliente o Amministratore).

La nostra visione di interfaccia utente per questo software è la seguente:

* L’Utente dialoga con una **Console**, un terminale (testuale o grafico) che mette in comunicazione Utente e Sistema PC Ready.
* Più Utenti devono poter accedere contemporaneamente al Sistema, per cui più istanze di Console possono esistere contemporaneamente.
* La Console è specializzata in base all’Utente che la sta utilizzando.
* La Console è un “dispatcher di eventi”, ovvero un oggetto che – al pari di un robot – ha accesso a diverse azioni disponibili, che può chiamare in qualunque momento.
* Tali azioni vengono chiamate **Comandi**.
* Ciascun Comando ha accesso al Sistema PC Ready, è può quindi richiedergli di svolgere determinate funzioni su handler e oggetti su cui abbia una determinata responsabilità.
* Il Comando può essere “interno” (o nascosto), e svolgerà quindi sole funzioni di utility per altri Comandi, oppure può essere accessibile all’Utente.
* La lista di Comandi accessibili dall’Utente è gestita dalla classe **ElencoComandi**, che si occupa di riconoscere anche quali Comandi siano destinati ai soli Clienti, quali ai soli Amministratori, e quali ad entrambi.

Con questa visione siamo sicuri di poter mostrare al cliente un prodotto funzionante fin da subito, e che possa in futuro prendere qualunque specifica forma il cliente desideri, in base all’evoluzione dei suoi requisiti.

**2. La classe Comando**

**Comando** è una classe “astratta” che implementa tre campi:

* Un **id** numerico, tale da poterlo distinguere univocamente dagli altri Comandi
* Una **descrizione** generica, da mostrare all’Utente tramite la Console, che spieghi in poche righe il compito da esso svolto
* Un campo booleano “adminOnly”, che – intuitivamente – permette alla Console di comprendere se un Utente di tipo Cliente abbia o meno diritto ad usufruire del Comando

Un Comando accessibile dall’Utente dovrà essere inizializzato fornendo tutti e tre i campi, mentre un Comando *“interno”* potrà essere inizializzato con dei valori di default.

Mentre tutte le sue funzioni pre-definite sono costituite dai getters/setters dei sopracitati campi, il suo *“cuore”* è la funzione astratta **“esegui”**, che prende come parametro un solo oggetto di tipo Console.

Ciascuno specifico Comando andrà definito creando una nuova classe a sé stante, che erediti dalla classe *“Comando”*, e che in quanto tale sia costretta a fornire una propria, esclusiva definizione della funzione *“esegui”*.

Si prenda – a titolo d’esempio – la classe “**ComandoSpegniConsole”**: tale Comando verrà chiamato quando l’Utente avrà intenzione di terminare le operazioni sul software, e verrà scelto dopo averne letto la descrizione *“Spegni la Console”*. Il corpo della sua funzione “esegui” si limita a fornire un messaggio d’addio all’Utente (“Grazie per aver usato PC Ready!”) e a chiamare le funzioni di spegnimento, sul Sistema – per garantire la permanenza dei nuovi dati – e sulla Console – per evitare che venga sfruttata da un Utente non autorizzato.

Inoltre, implementare ciascun Comando come una classe a sé stante anziché come una singola istanza di “Comando”, fornisce due vantaggi non indifferenti:

* Il programmatore ha la possibilità di definire le sue funzioni di utility da richiamare all’interno di “esegui”, per migliorare le operazioni di *testing*, *debugging* e *condivisione* e *lettura* del codice.
* L’aggiunta di un nuovo Comando equivale all’aggiunta di un nuovo file di classe nel progetto, richiedendo modifiche alla sola classe *ElencoComandi* e non a file critici quali le classi *“Console”*, *“PCReady”* o *“main”*.

**3. La classe ElencoComandi**

La classe **ElencoComandi** si occupa di fornire alla classe Console un’interfaccia sugli specifici Comandi che andrebbero resi disponibili all’Utente, nascondendo al contempo l’esistenza dei Comandi “interni”.

Per far ciò, contiene due attributi *“static”* pari a due array di Comandi, una per i soli Clienti (**elencoCliente**) e una per i soli Amministratori (**elencoAmministratore**).

Le quattro funzioni sono – rispettivamente – due per ottenere uno specifico Comando dall’elenco, dato il suo id interno (**getComandoCliente** e **getComandoAmministratore**) e due per ottenere una forma testuale dell’elenco di Comandi disponibili (**stringCliente** e **stringAmministratore**).

**4. La classe Console**

La classe Console è il fulcro principale dell’interattività fra Utente e Sistema, quella che implementa l’interfaccia utente a tutti gli effetti.

I suoi campi sono:

* **Sistema**: mantiene l’istanza singleton del Sistema PCReady per poterla sfruttare in qualunque momento.
* **On**: un booleano che determina se la Console è ancora utilizzabile, o se è stata “spenta” dall’Utente.
* **Admin**: un booleano che determina se l’Utente che ha accesso alla Console è un Amministratore o un regolare Cliente.
* **amministratoreCorrente**, **clienteCorrente**: due campi per tenere conto dell’istanza di Cliente/Amministratore a cui si sta fornendo l’accesso. Uno solo dei due campi viene inizializzato per ciascuna Console, alternativamente.

In aggiunta, è presente anche un campo di utility, lo **Scanner “in”**, poiché è richiesto per poter sfruttare le funzioni di input della libreria standard del linguaggio Java (per quelle di output è sufficiente accedere a System.out).

**4.1. Le funzioni di logica**

La logica interna di ciascuna Console è gestita tramite due sole funzioni, *“accesso”* e *“esegui”*:

* **“accesso”** si occupa di svolgere, prima di qualunque altro, il compito del Comando **“ComandoLogin”**, ossia quello di autenticare l’Utente come Cliente o Amministratore, e in assenza di credenziali pre-esistenti di farlo registrare (tramite il Comando **“ComandoRegistrazione”**).  
    
  Una volta avvenuta una delle due operazioni con successo, la Console conoscerà il suo Cliente/Amministratore corrente, e potrà procedere all’esecuzione degli altri Comandi disponibili: quindi viene chiamata la funzione “esegui”.
* **“esegui”** si occupa di mostrare all’Utente la lista di Comandi a sua disposizione (grazie a quanto detto precedentemente per la classe ElencoComandi, qui utilizzata). Una volta immesso, da parte dell’Utente, un codice valido fra quelli a sua disposizione, verrà chiamata la funzione “esegui” della corrispondente istanza di Comando.  
    
  Tale operazione è svolta ciclicamente fintanto che l’utente non chiama il Comando *“ComandoSpegniConsole”*.

**4.2. Le funzioni di utility**

La Console contiene al suo interno una famiglia di funzioni di Utility che i vari Comandi possono sfruttare per interagire con l’Utente. Poiché l’implementazione attuale della Console è sotto la forma di interfaccia testuale, tali funzioni si limitano ad effettuare un “wrapper” sopra le funzioni classiche di *input* (classe Scanner) e di *output* (System.out) fornite dal linguaggio.

La funzione **“print”** si limita – banalmente – a mostrare a schermo un messaggio testuale.

La famiglia di funzioni dal nome generico *“getTipo”* si occupa di valutare l’input testuale dell’Utente come una variabile dello specifico Tipo scelto, ed eventualmente di richiedere un nuovo input qualora quello analizzato non sia valido, mostrando uno specifico messaggio d’errore.

Ciascuna di queste funzioni presenta una variante (**override**) che accetta un parametro testuale in ingresso: tale stringa viene utilizzata come richiesta da mostrare a schermo all’utente per richiedere l’input, anziché stampare una richiesta generica (i.e. *“Inserire un numero intero”* nel caso della funzione **“getInt”**, che con l’override può essere sostituito da un messaggio custom).

Di queste funzioni, l’unica di particolare interesse è **“getYesNo”**, che trasporta il linguaggio informatico del tipo Booleano (true / false) nel linguaggio umano delle domande a risposta Sì/No, e che quindi implementa una logica leggermente più specifica per valutare l’input umano dell’Utente.

Le rimanenti funzioni della classe Console sono composte da getters e setters auto-generati.