Problema classe Customer / classe Operator

Ecco diverse soluzioni o idee per risolvere il problema della classe Utente (e in particolar modo delle sue potenziali classi figlie Cliente e Operatore). Notate che la soluzione deve essere la più estendibile, la più coerente, la più coesa e quella con più separazione di interessi (e bonus: quella con il codice migliore).

1)

**Unica classe User con tutto dentro**

PRO:  
- facile da implementare  
- il database ha una sola HashMap di tipo User dove ogni oggetto ha accesso ai propri metodi utili.

CONTRO:  
- un booleano per controllare lo stato cliente/operatore è molto debole  
- poco estendibile  
- diventa molto ingombrante subito.

2)

**Classe padre Operatore**

PRO:  
- permette alla classe figlia di specializzarsi, dato che l’operatore non ha bisogno di campi quali la data di iscrizione o la data di scadenza.

CONTRO:  
- la classe figlia Cliente vede ed eredita i metodi della classe padre, quindi l’utente potrebbe potenzialmente avere la possibilità di vedere l’elenco di tutti gli utenti iscritti *(e tutte le altre funzionalità future riservate solamente all’operatore).*

**Classe figlia Cliente**

PRO:  
- si specializza bene, con campi propri come la data di iscrizione e scadenza e i metodi per controllare la validità dell’iscrizione.

CONTRO:  
- eredita funzionalità non sue.

3)

**Classe padre Cliente**

PRO:  
- gli operatori sono “casi speciali” di clienti e hanno metodi che i clienti non possono avere.

CONTRO:  
- la classe figlia Operatore vede ed eredita i metodi della classe padre, quindi l’operatore ha a disposizione metodi e campi di cui non ha bisogno, quali il controllo della maggiore età, della scadenza, il rinnovo, ecc.

**Classe figlia Operatore**

PRO:  
- si specializza bene, soprattutto nelle release future.

CONTRO:  
- eredita funzionalità inutili e fastidiose.

4)

**Classe padre User**

PRO:  
- accomuna tantissimi campi *(nome, cognome, username, password, data di nascita)* e tanti metodi getter e setter delle classi figlie, permettendo di snellire il codice.

CONTRO:  
- la HashMap nel database contiene oggetti di tipo User, ma le classi figlie si specializzano e hanno metodi tutti loro, invisibili a User.   
Quando il database viene interrogato per ottenere informazioni *(ad esempio sui clienti)*, gli oggetti al suo interno non possono accedere ai metodi desiderati *(vedere quanto manca alla scadenza, filtrare gli utenti per maggiore età, ecc.)*

**Classe figlia Cliente**

PRO:  
- ottima specializzazione, ottima estendibilità.

CONTRO:  
- il database non contiene oggetti di tipo Cliente, quindi i metodi del cliente sono invisibili al database  
- per risolvere il problema sopra descritto, bisognerebbe spostare i campi di data iscrizione, data scadenza, maggiore età nella classe padre User, ma ciò renderebbe questa classe abbastanza inutile e ridondante e molto simile ad Operatore, per ora  
- oppure bisognerebbe fare ogni volta un cast all’oggetto desiderato, es:  
((Customer)user).nomeMetodoCliente();  
Ma questo lancia ClassCastException ogni volta che nel database si incontra un oggetto di tipo Operator, perché ovviamente sto castando l’oggetto user a Customer nonostante user sia di tipo Operator.   
Quindi bisognerebbe OGNI VOLTA controllare se è effettivamente istanza di Customer attraverso **instanceof**, poco elegante e troppo ingombrante.

**Classe figlia Operatore**

PRO:  
- ottima specializzazione, ottima estendibilità.

CONTRO:  
- abbastanza vuota e inutile per ora  
- stesso problema descritto a sinistra.

5)

**Interfaccia User**

PRO:  
- professional.

CONTRO:  
- troppo da sburroni.

**Classe Cliente**

PRO:  
- ottima specializzazione, ottima estendibilità.

CONTRO:  
- il database ora non ha più HashMap di un tipo solo User, bensì deve gestire sia HashMap Cliente che HashMap Operatore, similmente al problema (6).

**Classe Operatore**

PRO:  
- ottima specializzazione, ottima estendibilità.

CONTRO:  
- abbastanza vuota e inutile per ora.

6)

**Classe Cliente**

PRO:  
- ottima specializzazione, ottima estendibilità.

CONTRO:  
- staccata dalla classe Operatore ma con tanti campi simili/in comune

**Classe Operatore**

PRO:  
- ottima specializzazione, ottima estendibilità.

CONTRO:  
- staccata dalla classe Cliente ma con tanti campi simili/in comune

**Classe Database con una modifica fastidiosa**

**private** HashMap<String, Customer> customerList = **new** HashMap<>();  
**private** HashMap<String, Operator> operatorList = **new** HashMap<>();

**public** **void** addCustomer(Customer customer) { … }  
**public** **void** addOperator(Operator operator) { … }  
**public** **void** removeCustomer(Customer customer) { … }  
**public** **void** removeOperator(Operator operator) { … }  
**public** Customer fetchCustomer(Customer customer) { … }  
**public** Operator fetchOperator(Operator operator) { … }  
**public** **boolean** isPresent(Customer customer) { … }   
**public** **boolean** isPresent(Operator operator) { … }   
**public** HashMap<String, Customer> getCustomers() { … }   
**public** HashMap<String, Operator> getOperators() { … }

CONTRO:  
- ripetizione di strutture dati, metodi, campi. Bruttissimo da vedere, spezza in due le liste e permette di gestire liste diverse con oggetti diversi, ma è ingombrante e poco pratico.

7)

**Classe User con un enum che ha due stati, Cliente e Operatore**

PRO:  
- professionale e abbastanza pulito.

CONTRO:  
- difficile e contro-intuitivo da gestire.

8)