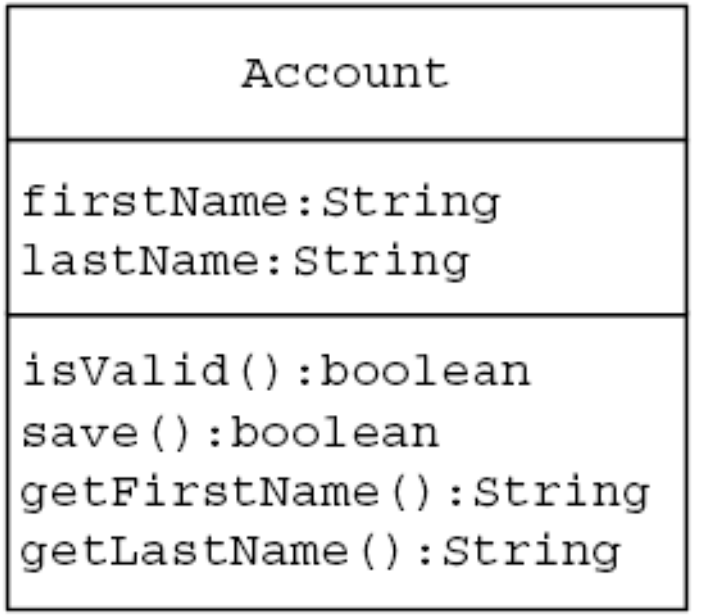
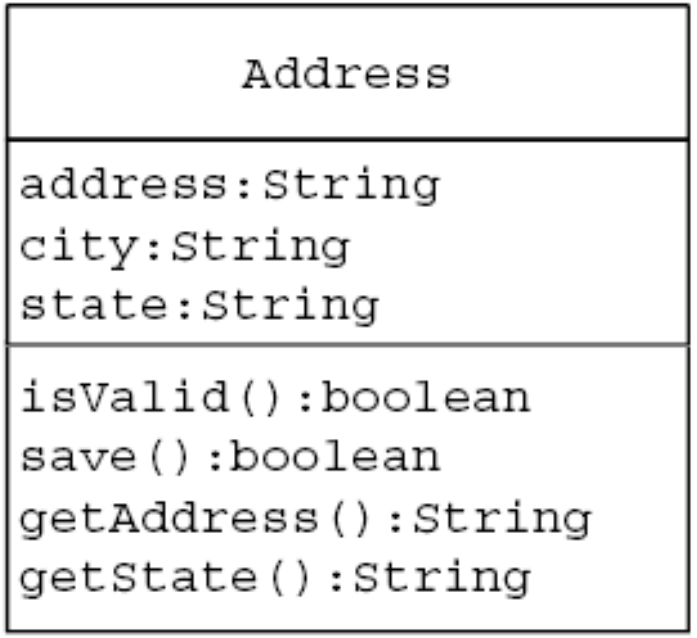
# El Problema

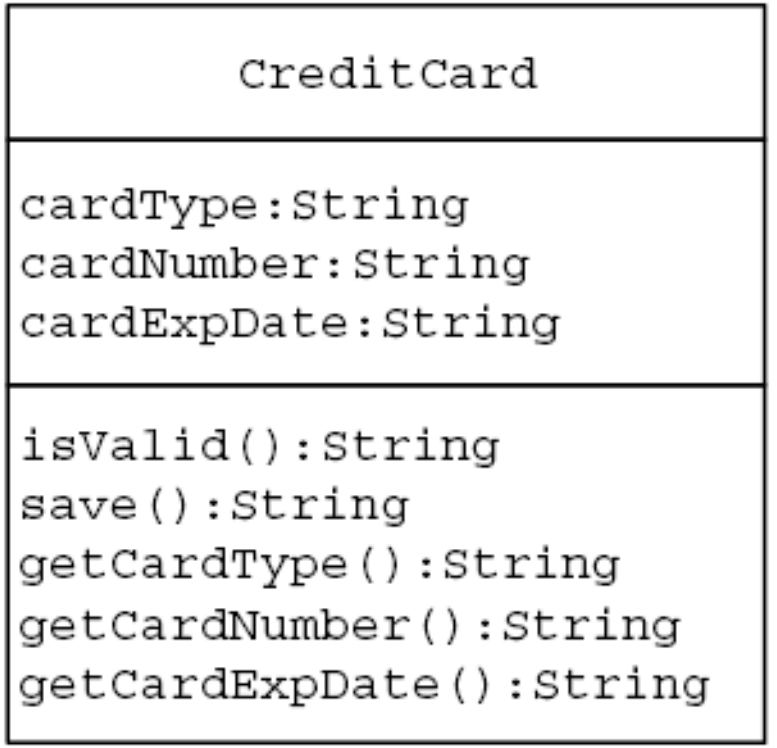
La classe client treballa sobre una familia d'objectes relacionats sobre els que es fan operacions.

Aquests canvis es guarden (en base dades).

L'aplicació pot (o no) estar conectada al servidor sense saber-ho



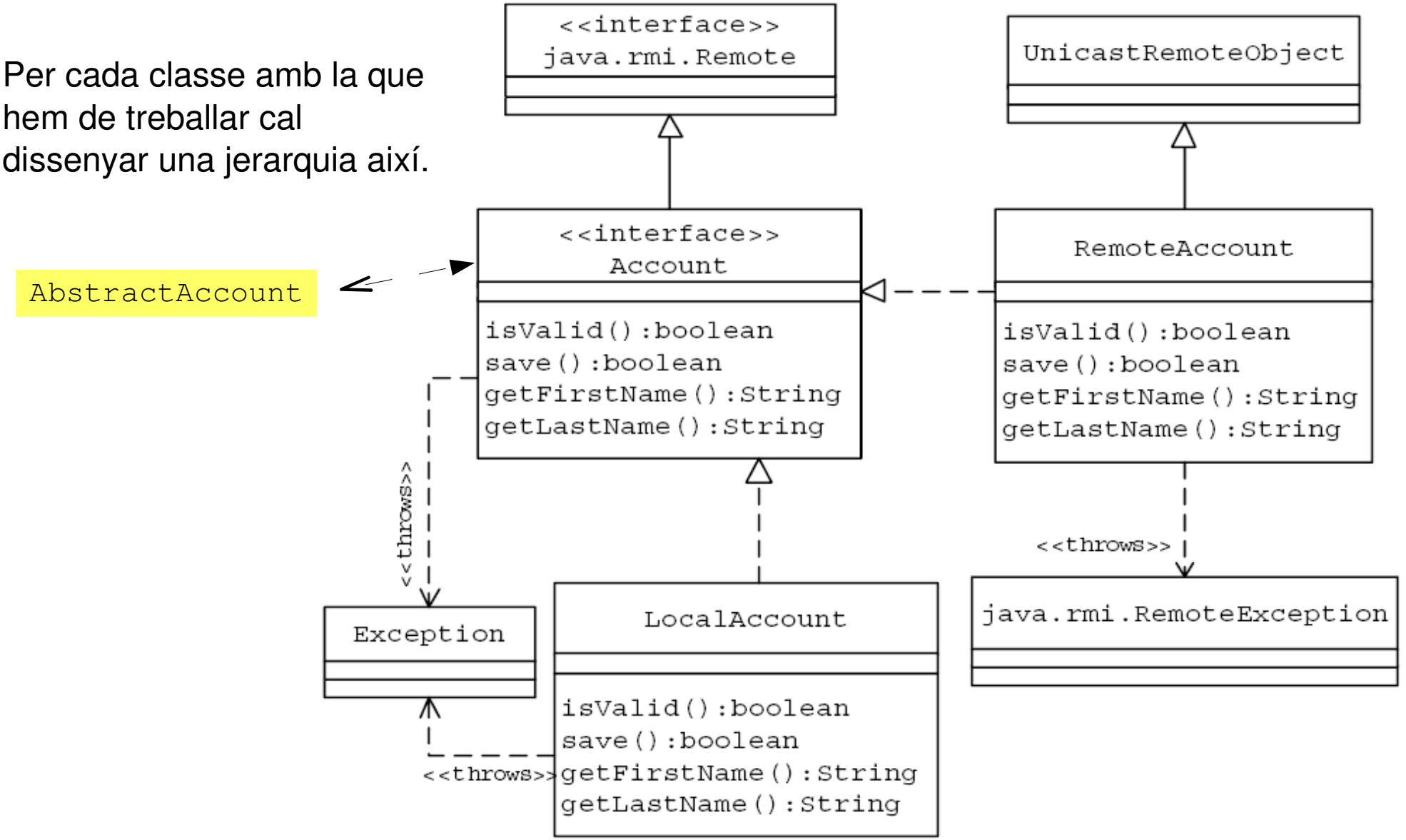



# Funcionament en local i funcionament en línia via RMI

- Extendre la classe `java.rmi.server.UnicastRemoteObject`
- Implementar la interfície `java.rmi.Remote`
- Els mètodes han de disparar l'excepció `java.rmi.RemoteException`

# Suport per a RMI

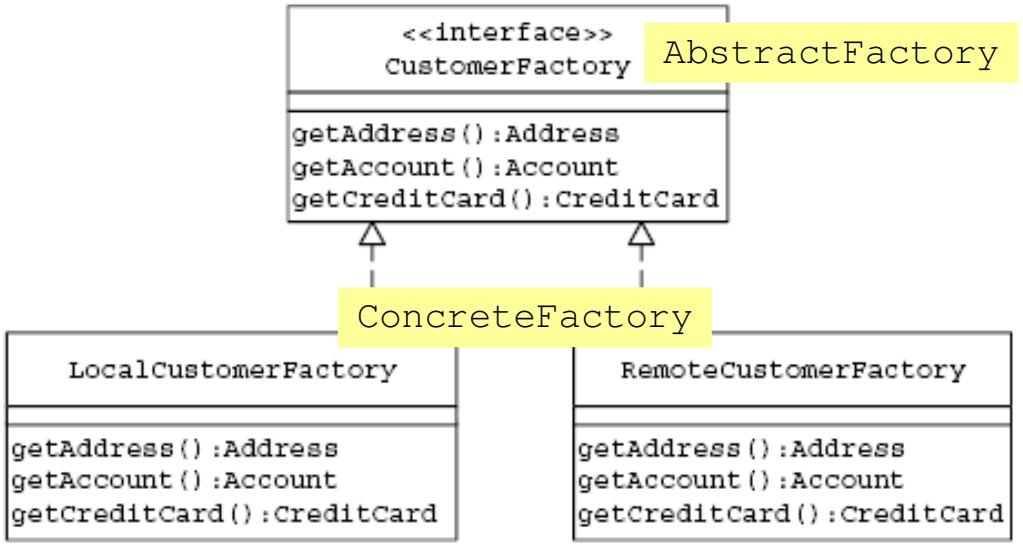
Per cada classe amb la que hem de treballar cal dissenyar una jerarquia així.

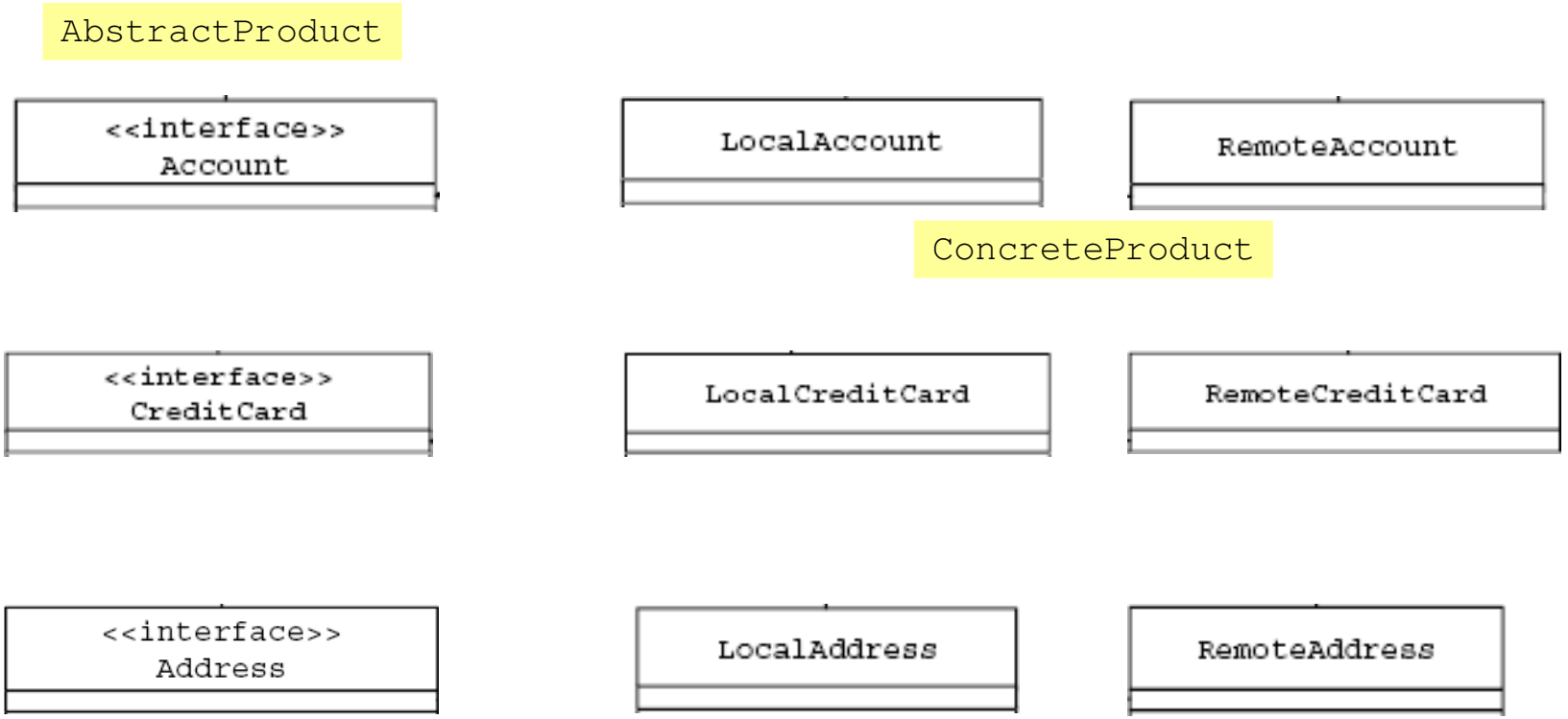


# Necessitat d'Abstract Factory

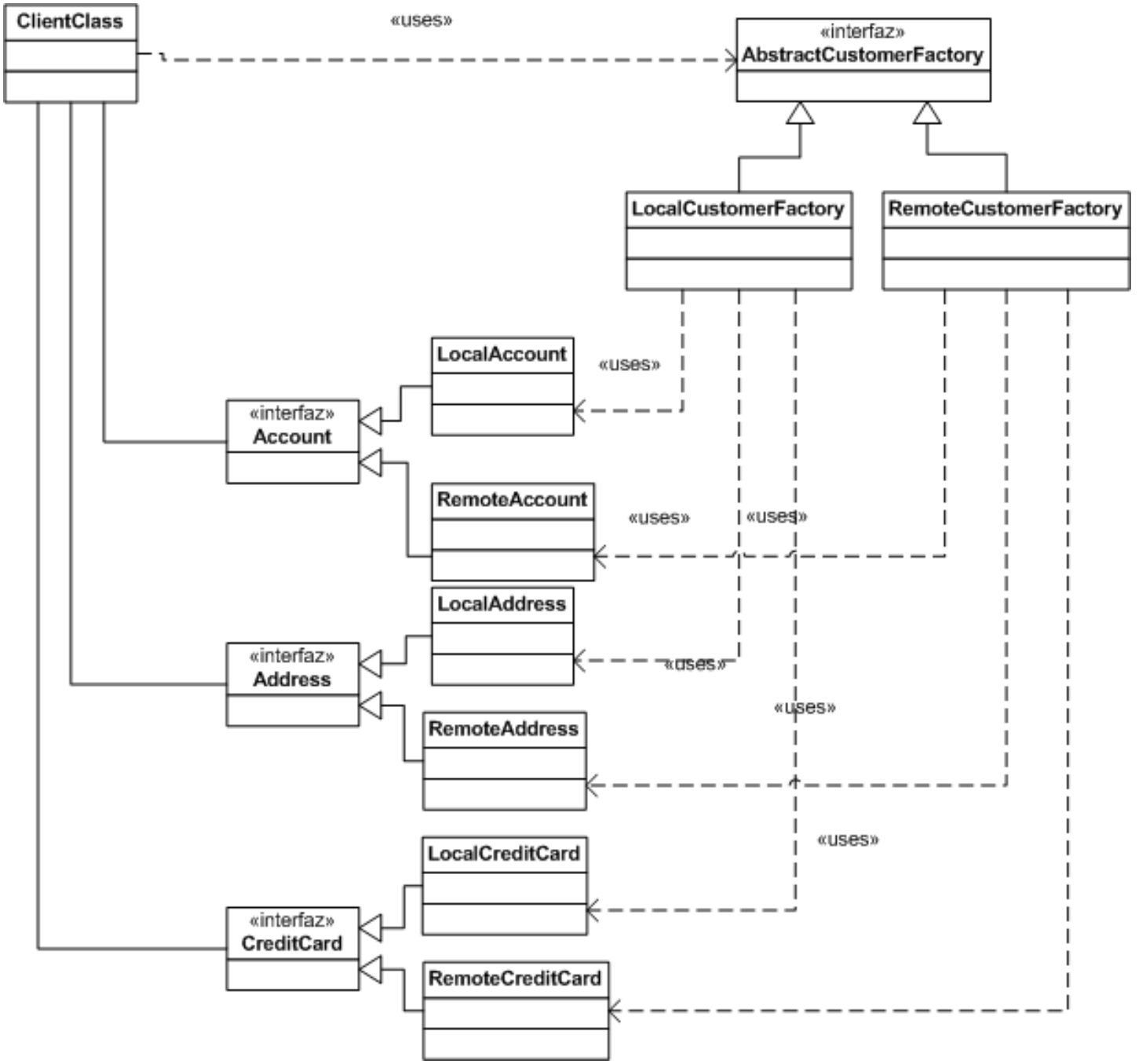
- Tenim una familia d'objectes de classes diferents:
  - `Account`
  - `CreditCard`
  - `Address`
- Cada objecte és de fet, una jerarquia amb diverses opcions (implementacions):
  - Treball en local
  - Treball remot via RMI
  - El més important:
    **La classe client desconeix el mode de treball, senzillament guarda objectes amb el mètode `save()`**

# Classes necessàries per aplicar el Patró



# Diagrama de classes resultant



# ¿Què guanyem?

- Escalabilitat i noves funcions amb el mateix esquema
- Podem afegir nous mètodes remots o nous elements dins `Customer`
- Amb el disseny adequat fins i tot podem fer que alguns elements funcionin en mode local i altres en remot

**Transparència i reusabilitat:**

Tot transparent, sense que la classe client `ClientClass` se n'adoni i sense fer-hi cap canvi

# Exemple

Volem permetre un altre mode de treball remot per guardar les dades remotament via XMLRPC

Cal:

- Implementar les classes XMLRPC per a `Account`, `Address` i `CreditCard` (s'hauria de fer de tota manera)
- Implementar la classe `XMLRPCCostumerFactory` per usar els noves classes
- Afegir la lògica de decissió del mode (Factoria) a instanciar

Resultat:

Les classes usuàries del nostre desenvolupament haurien guanyat una nova funcionalitat. Sense saber-ho i sense necessitat de modificar-les.