Corso di
“Sviluppo di applicazioni Web”

Esercitatore: Giovanni Grasso

- Hibernate: mapping di gerarchie di classi

Gerarchie di classi

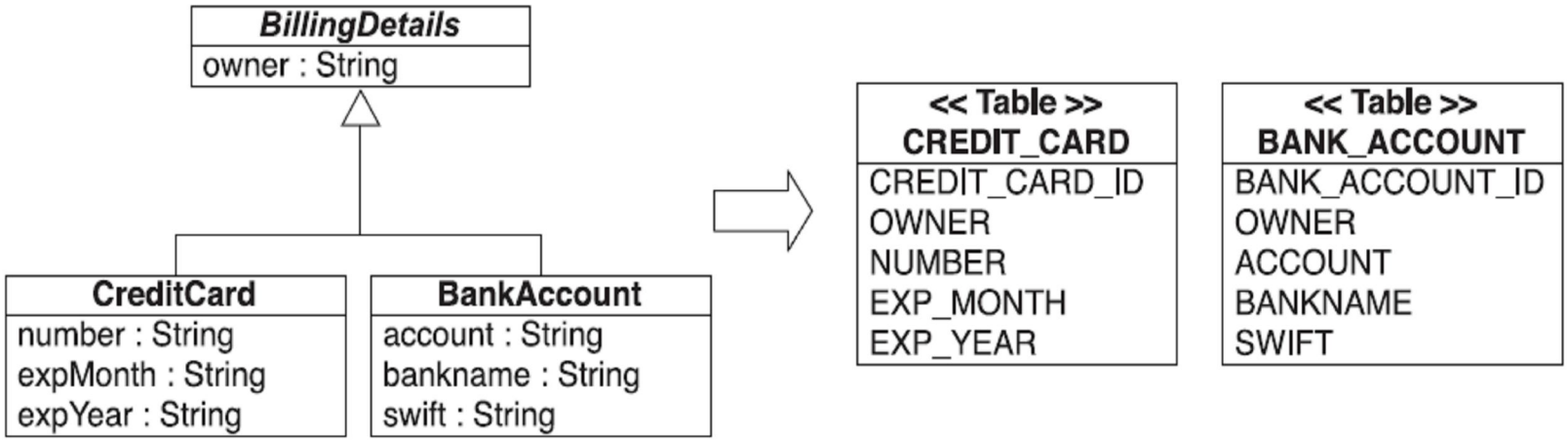
- Possono essere gestite con 4 approcci differenti:
  - Table per concrete class con mapping del polimorfismo
    - Non si asserisce nulla riguardo il mapping dell'ereditarietà
    - Ogni classe diventa una tabella a sè, le proprietà sono ripetute come attributi in ogni tabella, se indicate come persistenti
  - Table per concrete class
    - Polimorfismo ed ereditarietà non hanno nessun mapping nello schema SQL
  - Table per class hierarchy
    - Si usa uno schema SQL denormalizzato, che "riassume" tutta la gerarchia di classi in un'unica tabella
    - Una colonna funge da "discriminator" → quale delle classi di una gerarchia mappa la tupla?
  - Table per subclass
    - La relazione di ereditarietà *is-a* viene mappata come un *has-a*
    - La classe principale dispone di una tabella, la quale riferisce, per mezzo di relazioni di chiave esterna, altre tabelle che contengono gli attributi specifici delle sottoclassi

Gerarchie di classi

- La modalità di mapping si specifica con l'annotazione @Inheritance
  - È sempre necessario l'id nella superclasse
- InheritanceType.TABLE_PER_CLASS → "table per concrete class"
- InheritanceType.SINGLE_TABLE → "table-per-class-hierarchy"
  - @DiscriminatorColumn indica l'attributo "discriminator" (nella superclasse)
  - @DiscriminatorValue va usato, nelle sottoclassi, per attribuire un valore al discriminator che funga da contrassegno
  - Hibernate supporta anche la possibilità di desumere il discriminator per mezzo di una formula SQL
    - Utile per il mapping di schemi *legacy*
  - Si noti che, con questo approccio, i campi devono essere tutti *nullable*
- InheritanceType.JOINED → "table per subclass"
  - È anche possibile ottenere il comportamento equivalente solo per una sottoclasse, marcando quest'ultima con @SecondaryTable

Table per concrete class con mapping del polimorfismo

•Non si asserisce nulla riguardo il mapping dell'ereditarietà
•Ogni classe diventa una tabella a sè, le proprietà sono ripetute come attributi in ogni tabella, se indicate come persistenti



•Problemi: non supporta bene le associazioni polimorfe. Non basta una chiave esterna nella superclasse ma occorre in tutte le sottoclassi.
•Anche le query polimorfe sono un problema: richiede più queries sulle sottoclassi
•Duplicazione

```
@MappedSuperclass
public abstract class BillingDetails implements Serializable, Comparable {
    @Column(name = "OWNER", nullable = false)
    private String owner;
.....
@Entity
@AttributeOverride(name = "owner", column =
@Column(name = "CC_OWNER", nullable = false)
)
public class CreditCard extends BillingDetails {
@Id @GeneratedValue
@Column(name = "CREDIT_CARD_ID")
private Long id = null;
@Column(name = "NUMBER", nullable = false)
private String number;
```

Table per concrete class con unione

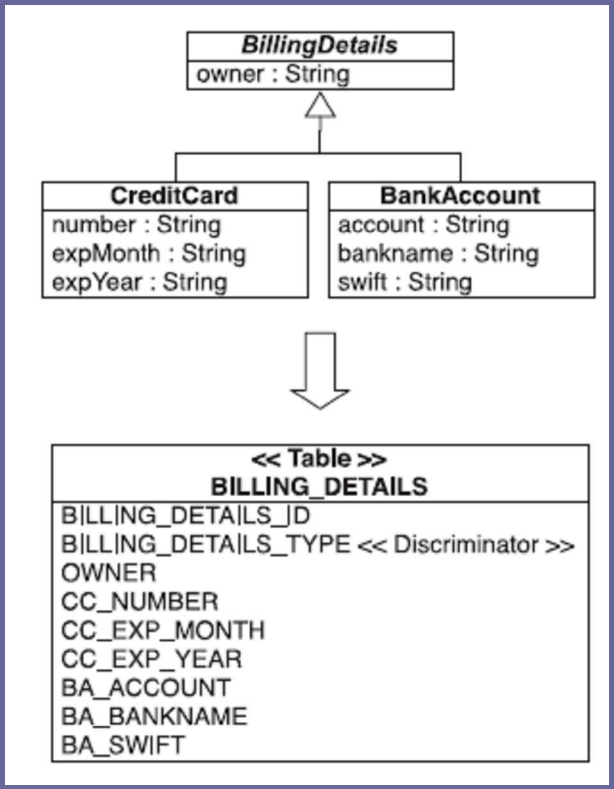
- Condivisione delle proprietà della superclasse, no duplicazione
- Associazioni possibili a livello di superclasse
- Hibernate risolve le query polimorfe usando Union

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class BillingDetails {
@Id @GeneratedValue
private Long id = null;
@Column(name = "OWNER", nullable = false)
private String owner;
...
}

@Entity
@Table(name = "CREDIT_CARD")
public class CreditCard extends BillingDetails {
@Column(name = "NUMBER", nullable = false)
private String number;
...
```

Table per class hierachy

- Si usa uno schema SQL denormalizzato, che “riassume” tutta la gerarchia di classi in un’unica tabella
- Una colonna funge da “discriminator” → quale delle classi di una gerarchia mappa la tupla?



Strategia semplice e performante

Ottima rappresentazione del polimorfismo

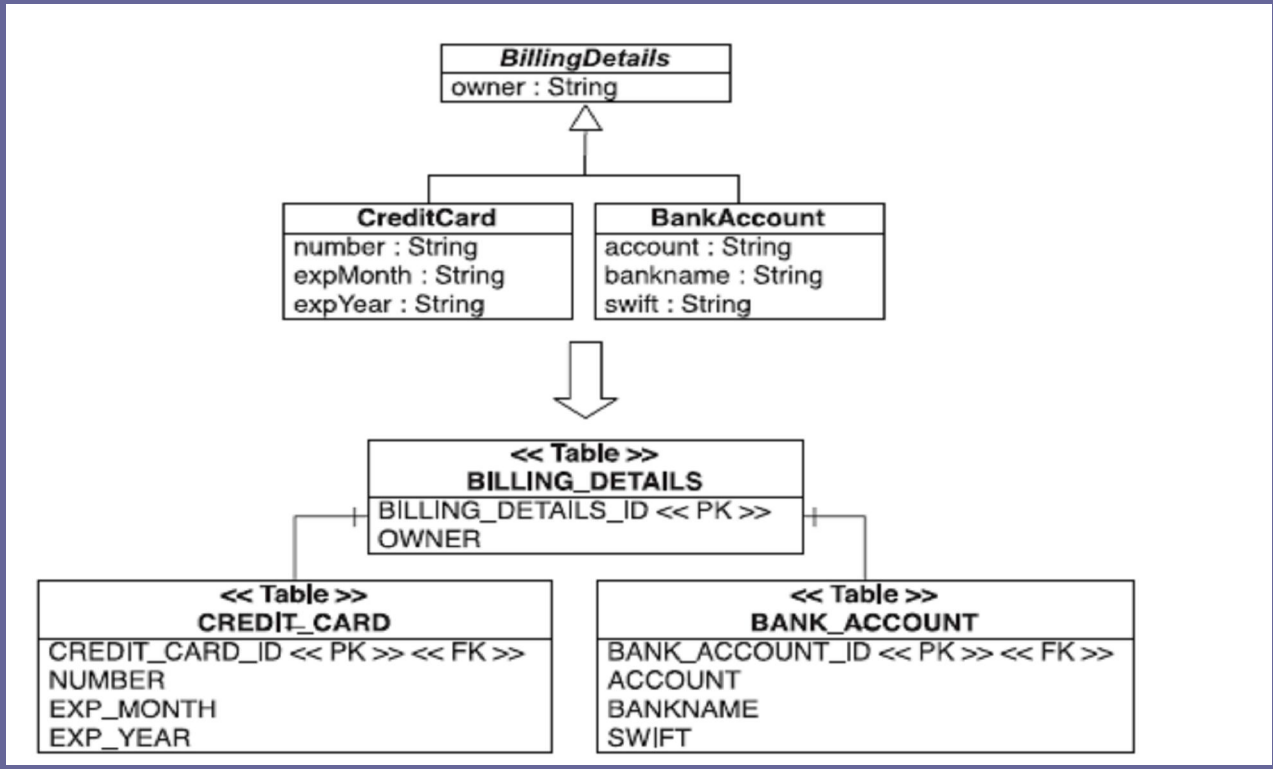
Problemi: colonne devono essere nullable, denormalizzazione dello schema

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(
name = "BILLING_DETAILS_TYPE",
discriminatorType = DiscriminatorType.STRING)
public abstract class BillingDetails {
@Id @GeneratedValue
@Column(name = "BILLING_DETAILS_ID")
private Long id = null;
@Column(name = "OWNER", nullable = false)
private String owner;
...}

@Entity
@DiscriminatorValue("CC")
public class CreditCard extends BillingDetails {
@Column(name = "CC_NUMBER")
private String number;
...
}
```

Table per subclass

La relazione di ereditarietà *is-a* viene mappata come un *has-a*
La classe principale dispone di una tabella, la quale riferisce, per mezzo di relazioni di chiave esterna, altre tabelle che contengono gli attributi specifici delle sottoclassi



Ogni classe/sottoclasse (anche astratta) che ha proprietà persistenti ha la sua tabella. (Non c'è duplicazione nelle sottoclassi)

Ogni proprietà di una sottoclasse è memorizzta insieme ad una primary key che è anche chiave esterna nella tabella della superclasse

- Vantaggi nella normalizzazione dei dati

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class BillingDetails {
@Id @GeneratedValue
@Column(name = "BILLING_DETAILS_ID")
private Long id = null;
...}
```

```
@Entity
public class BankAccount {
...
}
```

In subclasses, you don’t need to specify the join column if the primary key column of the subclass table has (or is supposed to have) the same name as the primary key column of the superclass table

BankAccount has no identifier property; it automatically inherits the BILLING_

- DETAILS_ID property and column from the superclass, and Hibernate knows how to join the tables together if you want to retrieve instances of BankAccount. Of
- course, you can specify the column name explicitly:
- @Entity
- @PrimaryKeyJoinColumn(name = “Bank_account_ID")