

# Object-Relational Mapping

Versione Preliminare

**Antonella Poggi**

Dipartimento di informatica e Sistemistica  
Sapienza Università di Roma

**Progetto di Applicazioni Software**  
**Anno accademico 2008-2009**

*Questi lucidi sono stati prodotti sulla base del materiale preparato per il corso di Progetto di Basi di Dati da D. Lembo e A. Calì.*

## Approccio Object-Oriented (OO)

---

- **Approccio Object-Oriented (OO)**, basato cioè sull'uso di linguaggi ad oggetti quali ad es. C++, Java, etc., è ad oggi l'approccio/paradigma più consolidato ed usato per la realizzazione di applicazioni
- Le metodologie di progettazione OO prevedono l'uso di linguaggi di modellazione concettuale, come **UML**, che offrono un supporto fondamentale per progettare gli strati di logica dell'applicazione e di interfaccia utente di un qualsiasi sistema informatico

## Approccio Object-Oriented (OO) e persistenza

---

- Un sistema informativo che non preserva dati al suo spegnimento è di poca utilità
- I linguaggi di programmazione OO non offrono soluzioni “native” al problema della persistenza, ovvero al problema della gestione di dati persistenti da parte di applicazioni OO

## DBMS relazionali

---

- I **DBMS relazionali** sono ad oggi la tecnologia più consolidata, efficiente e comunemente usata per la gestione di grandi quantità di dati
- Offrono solo qualche ausilio alla programmazione “procedurale” di applicazioni (e.g. stored procedure)
- Non sono specifici di alcun paradigma di programmazione, n in generale, di alcuna applicazione  
→ principio noto come **data independence**, ovvero *in generale, i dati vivono più a lungo di qualsiasi applicazione*

## **Binomio (applicazione OO - DBMS relazionale)**

---

- Necessità di risolvere il problema della persistenza in applicazione OO mediante l'uso di DBMS relazionali, ovvero di permettere alle applicazioni di gestire dati memorizzati in memoria secondaria in un DBMS relazionale
- Problema: il paradigma OO si è sviluppato su principi propri dell'ingegneria del software, quello relazionale affonda le radici su principi matematici, e le differenze fra i due risultano considerevoli

## Impedance mismatch tra modello OO e relazionale

---

- **Impedance mismatch** è un termine preso in prestito dall'ingegneria elettrica per denotare la mancata corrispondenza tra modello OO e relazionale
  - Modello OO: descrive il dominio in termini di **oggetti** e loro **proprietà**, che possono essere **attributi** (i.e. **valori**) o **associazioni** ad altri oggetti
  - Modello relazionale: descrive il dominio in termini **relazioni** tra **valori**

- L'implementazione di un'applicazione ad oggetti è basata sull'uso di un linguaggio di programmazione OO  
≠ l'implementazione di una base di dati è basata sull'uso di un linguaggio relazionale (l'algebra)  
⇒ abbiamo il problema di far colloquiare l'applicazione che parla in termini di *oggetti* con un DBMS che parla in termini di *valori*  
⇒ per far coesistere applicazioni ad oggetti e DBMS relazionali occorre uno sforzo sia progettuale che implementativo

## Differenze fondamentali

---

1. **Coesione** In un oggetto, tutte le sue proprietà sono contenute nell'oggetto medesimo  
 $\neq$  i dati relazionali corrispondenti ad una stessa entità possono essere stati splittati in più tabelle, e.g. a seguito della fase di ristrutturazione dello schema logico
2. **Incapsulamento** In un oggetto, la “business logic” risiede (parzialmente) nei metodi dell'oggetto stesso  
 $\neq$  i dati relazionali e la logica che li governa sono implementati in maniera separata

3. **Granularità dei tipi di dato** Tipi di dati composti sono tipicamente rappresentati nei linguaggi OO mediante apposite classi di oggetti con loro proprietà, mentre l'SQL non prevede alcun meccanismo standard per la definizione di tipi di dato composti  
→ la classe **Indirizzo** farà tipicamente parte delle classi di dominio e qualsiasi oggetto con proprietà di tipo **Indirizzo** parteciperà ad un'associazione con essa, mentre le informazioni che compongono un indirizzo quali la via, il numero civico e il CAP saranno loro stesse tipicamente attributi dell'entità avente come proprietà il corrispondente indirizzo

4. **Ereditarietà** e **Polimorfismo** Nei linguaggi OO  
l'ereditarietà è implementata attraverso l'uso di super-  
e sotto-classi, e fornisce inoltre la possibilità del  
polimorfismo  
≠ il modello relazionale non ha la possibilità di  
rappresentare ereditarietà e associazioni polimorfiche:  
in particolare, l'ereditarietà è “simulata” attraverso la  
replicazione di dati e l'uso di vincoli di integrità, mentre  
non esiste alcun meccanismo per simulare il  
polimorfismo

5. **Identità** Nei linguaggi OO esistono due nozioni di identità: identità *fisica* di oggetti, ovvero di locazioni di memoria in cui gli oggetti sono memorizzati, e identità *semantica*, implementata attraverso la definizione del metodo **equals**  
≠ l'identità di una tupla è implementata attraverso l'uso della chiave primaria e non corrisponde direttamente a nessuna delle due nozioni di cui sopra
6. **Navigabilità** Nei linguaggi OO il dominio “si naviga” da un oggetto all'altro, attraverso le associazioni, ovvero secondo la responsabilità delle classi che partecipano all'associazione (**Ricorda:** a partire da un oggetto si possono accedere gli oggetti con cui partecipa ad una associazione solo se ha responsabilità su di essa)  
≠ l'accesso ai dati relazionali avviene mediante l'uso di join tra tabelle

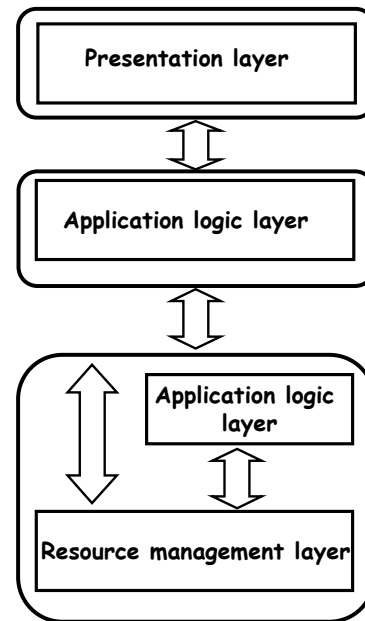
## Object-relational mapping

---

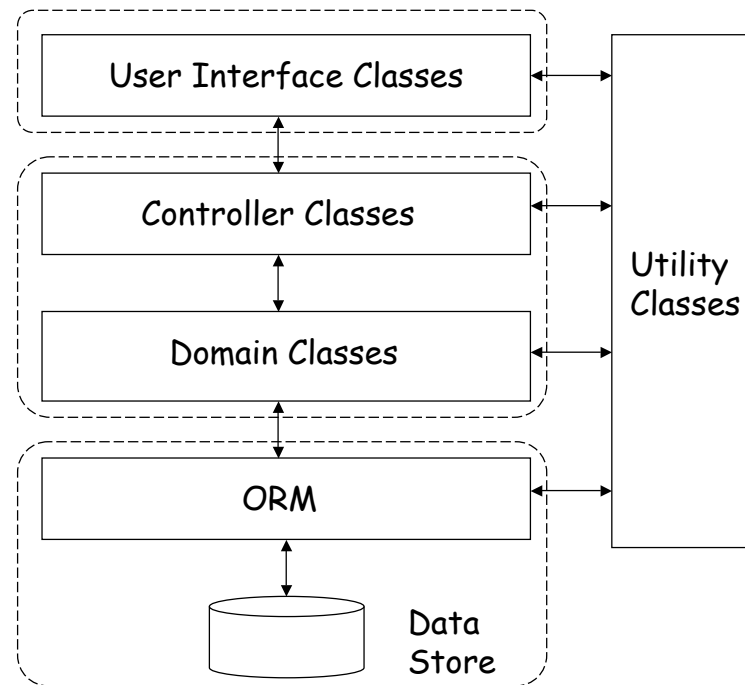
- La soluzione al problema della persistenza richiede un meccanismo per specificare la corrispondenza tra il dominio dell'applicazione e la base di dati che risolva opportunamente il problema dell'impedance mismatch
- Tale meccanismo è comunemente chiamato **object-relational mapping (ORM)**
- Idea: si identificano le classi di oggetti dell'applicazione di tipo **persistente** e si fa in modo che i loro attributi e proprietà siano "mappati" su dati memorizzati in una base di dati relazionale
- Su tali oggetti saranno definite ed invocate operazioni di creazione, modifica, cancellazione, con l'obiettivo di effettuare letture e modifiche sulla base dati sottostante

## Richiami: architetture a 3 livelli per le applicazioni

---



## Architettura in termini di applicazioni OO e DBMS relazionali



## Architettura dell'applicazione

---

**Persistence classes** gestiscono l'accesso ai dati, incapsulando le modalità di accesso

**Domain classes (Business Objects)** Sono le classi che rappresentano i concetti pertinenti al dominio dell'applicazione. Ogni classe deve incorporare il comportamento specifico delle sue istanze.

**Controller Classes** Le classi di controllo incorporano la logica di business. Le operazioni implementate in una classe di controllo possono coinvolgere più di una domain class ed eventualmente altre classi di controllo.

## Architettura dell'applicazione

---

**User Interface Classes** Sono le classi che incorporano l'interfaccia offerta direttamente all'utente.

**Utility Classes** Ogni applicazione ha un insieme infrastrutturale di classi, chiamate Utility classes, che sono usate ad ogni livello dell'applicazione (e.g. Class Exception)

## Diverse strategie ORM

---

Strategie possibili:

1. Forza bruta
2. Codifica manuale di uno strato per l'ORM  
→ oggetti per l'accesso ai dati (Data Access Objects, DAO)
3. Persistence framework

## ORM: forza bruta

---

- Prevede di equipaggiare le classi dell'applicazione (in particolare le classi di dominio) con metodi che interagiscono direttamente con la base di dati, "ovunque" necessario, ovvero incorpora la logica di accesso ai dati sul DB **nelle domain classes** e **nelle controller classes**
- È ragionevole quando l'applicazione è sufficientemente semplice e si può fare a meno di uno strato di incapsulamento (**persistence classes**)

## ORM: forza bruta (cont.)

---

Un oggetto di una classe di business si popola con i dati presenti in un database (acceduto ad es. attraverso driver JDBC), procedendo come segue:

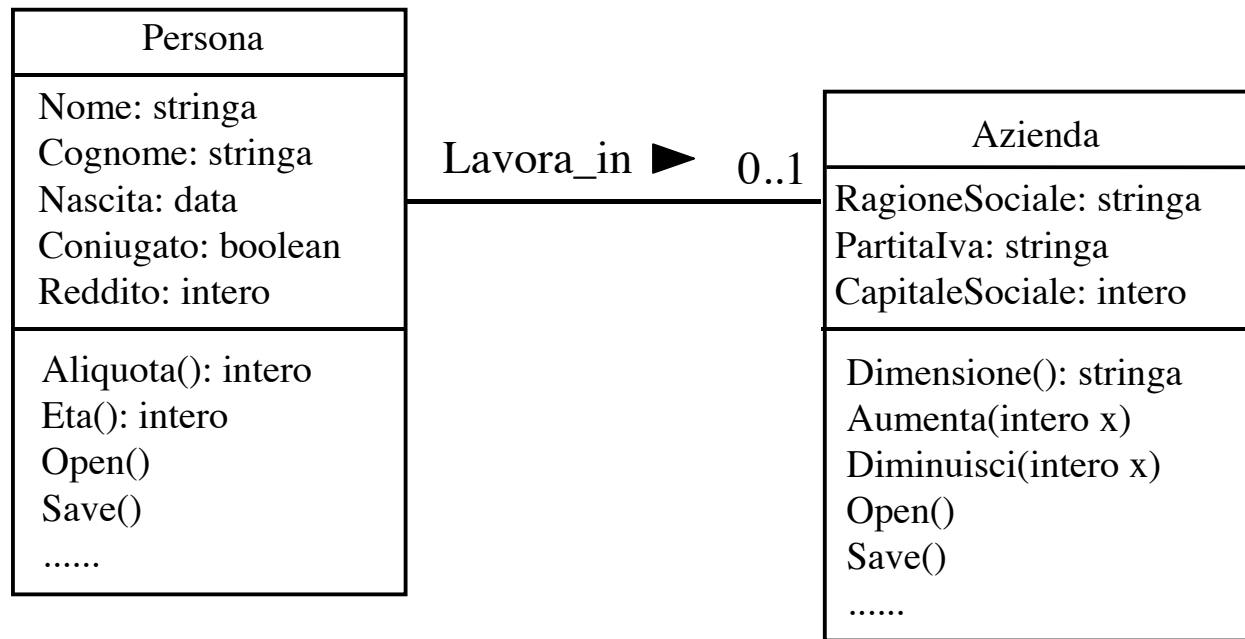
- costruisce da sé lo statement SQL necessario;
- lo passa al driver;
- riceve i risultati dal database;
- li elabora opportunamente.

## ORM: forza bruta - svantaggi

---

- Non è una strategia di incapsulamento
- Accoppia fortemente lo strato delle domain classes e dei controller al database
- Richiede che chi progetta l'applicazione abbia conoscenza dettagliata del database

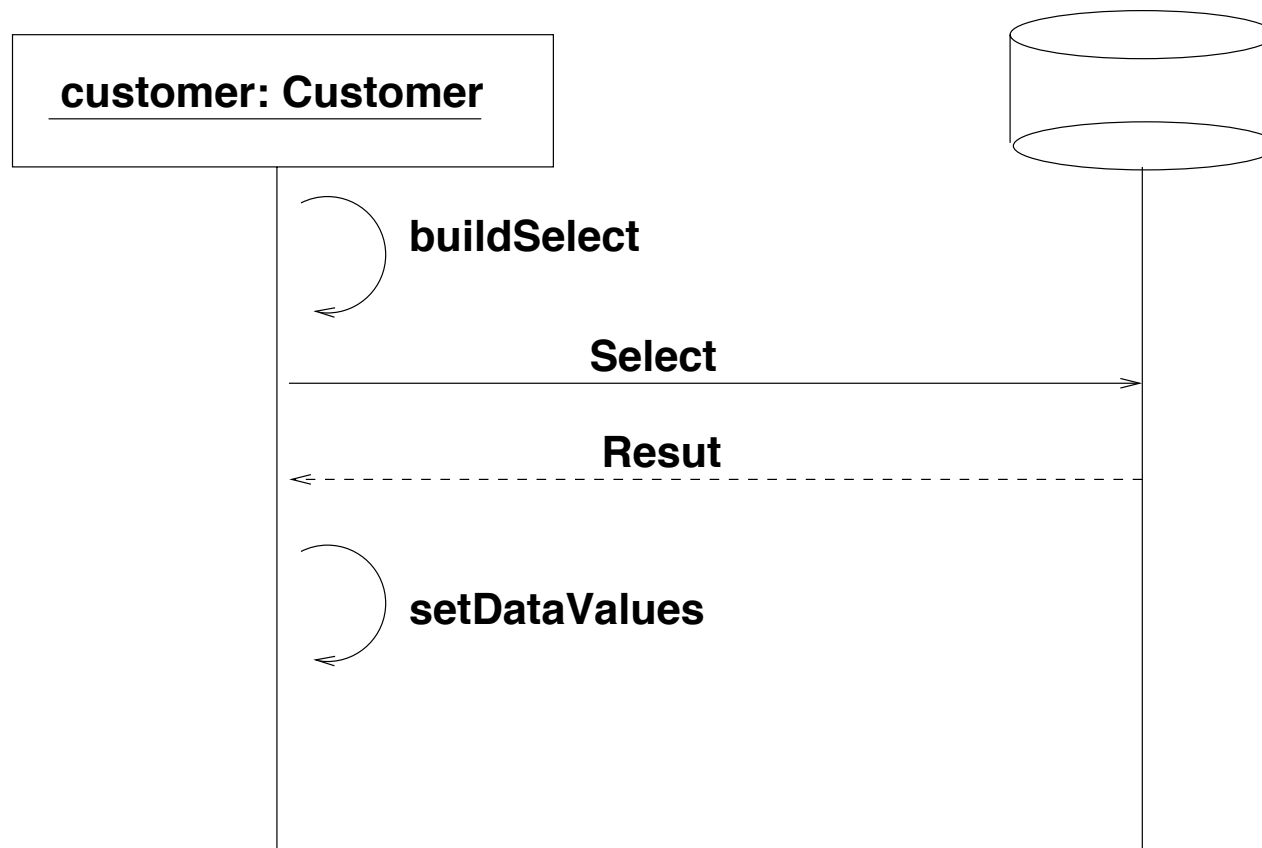
## ORM Forza bruta: esempio



Nelle classi che modellano oggetti del dominio, si potrebbe avere un metodo per popolare un oggetto con i dati del DB (`Open()`), o per rendere persistenti i cambiamenti effettuati sull'oggetto (`Save()`), etc.

## Forza bruta (cont.)

Leggere un singolo oggetto dal DB



## Forza Bruta

---

**Questo approccio è da evitare.** Infatti, in tal modo si offre un canale diretto dalla logica dell'applicazione al DB, che viola così:

- interfacciamento esplicito: non è definito in maniera chiara il passaggio di informazione fra l'applicazione ed il DB
- information hiding: non nascondo all'applicazione i dettagli della base dati

## ORM: DAO

---

- Prevede di realizzare uno strato dell'applicazione (chiamato appunto DAO) demandato completamente a gestire la comunicazione fra l'applicazione ed il DBMS
- anche in questo caso il mapping è realizzato manualmente attraverso l'uso di JDBC/SQL
- l'accesso al DB viene però opportunamente **incapsulato** nelle classi DAO:
  - nasconde alla logica di business codice JDBC complesso e SQL non-portabile
  - fornisce un interfacciamento esplicito del codice
  - migliora la modularità, risolve problemi di accoppiamento tipici dell'approccio forza bruta

## ORM: DAO (Cont.)

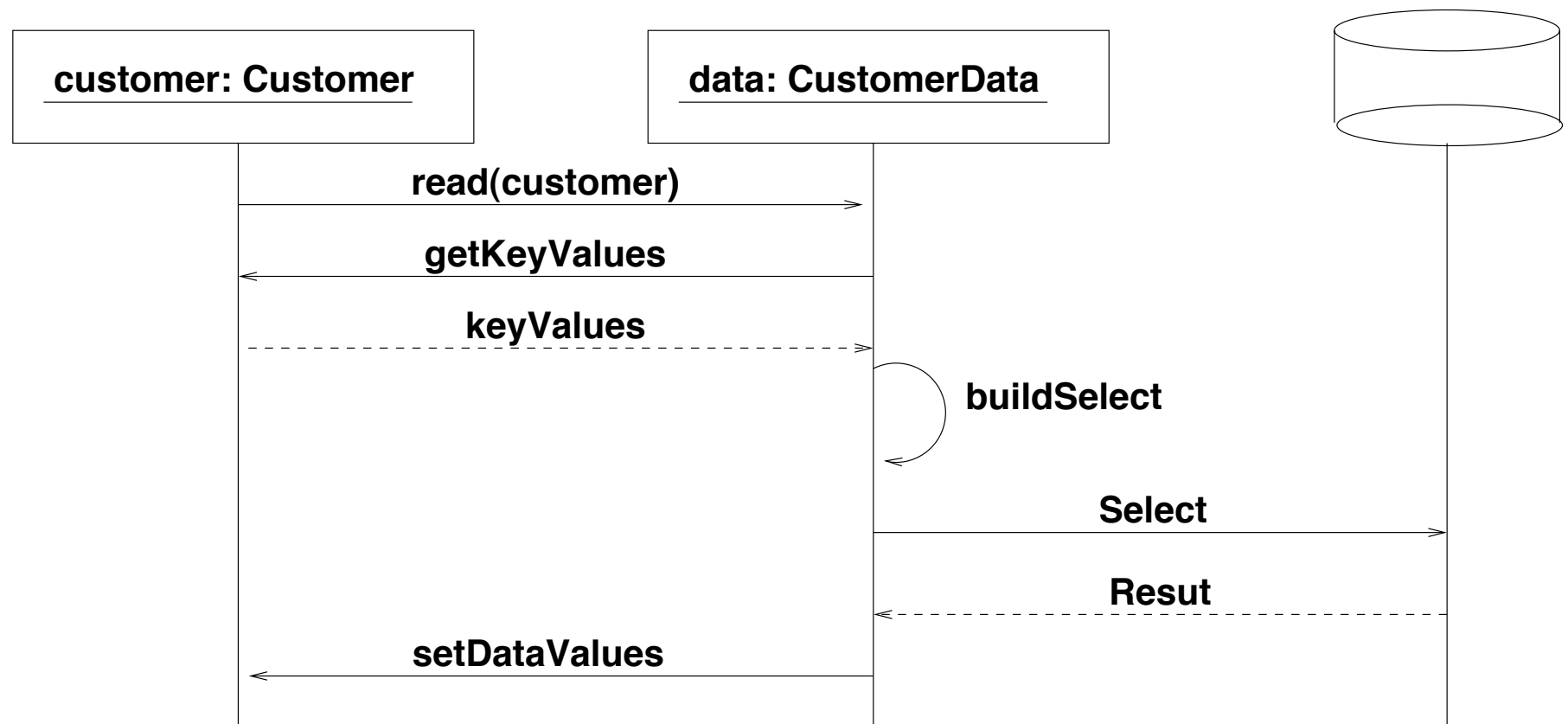
---

Un oggetto di una classe di business, per accedere al database

- invoca metodi di una classe demandata a gestire gli accessi al DB (relativamente alle informazioni richieste)
- è questa classe che costruisce lo statement SQL e lo passa al driver, riceve i risultati dal database e li inoltra alla classe di business che la ha interrogata
- la classe di business effettua poi le sue elaborazioni sui dati ricevuti

## ORM: DAO – esempio

Leggere un singolo oggetto dal DB



## ORM: DAO (cont.)

---

- Tutta la logica di accesso al DB è completamente incapsulata nelle classi DAO
- Cambiamenti del DB influenzano **solo** le DAO
- La classe CustomerData si fa carico di gestire il codice SQL, mentre tutto ciò è trasparente rispetto alla classe Customer (che è una **domain class**), la quale invoca metodi indipendenti dal DB
- L'approccio tipico è quello di avere un DAO per ciascuna domain class

## ORM: Persistence Framework

---

- prevede l'utilizzo di un framework predefinito (ad es. **Hibernate**) per la gestione della persistenza
- l'obiettivo è liberare il programmatore quanto più possibile dalla necessità di scrivere codice SQL nella sua applicazione.
- il codice SQL viene generato automaticamente sulla base di informazioni di meta-livello fornite dal programmatore (ad es. all'interno di file di configurazione)
- incapsulamento completo: il programmatore vede il DB solo quando configura il framework

## ORM: Persistence framework (cont.)

---

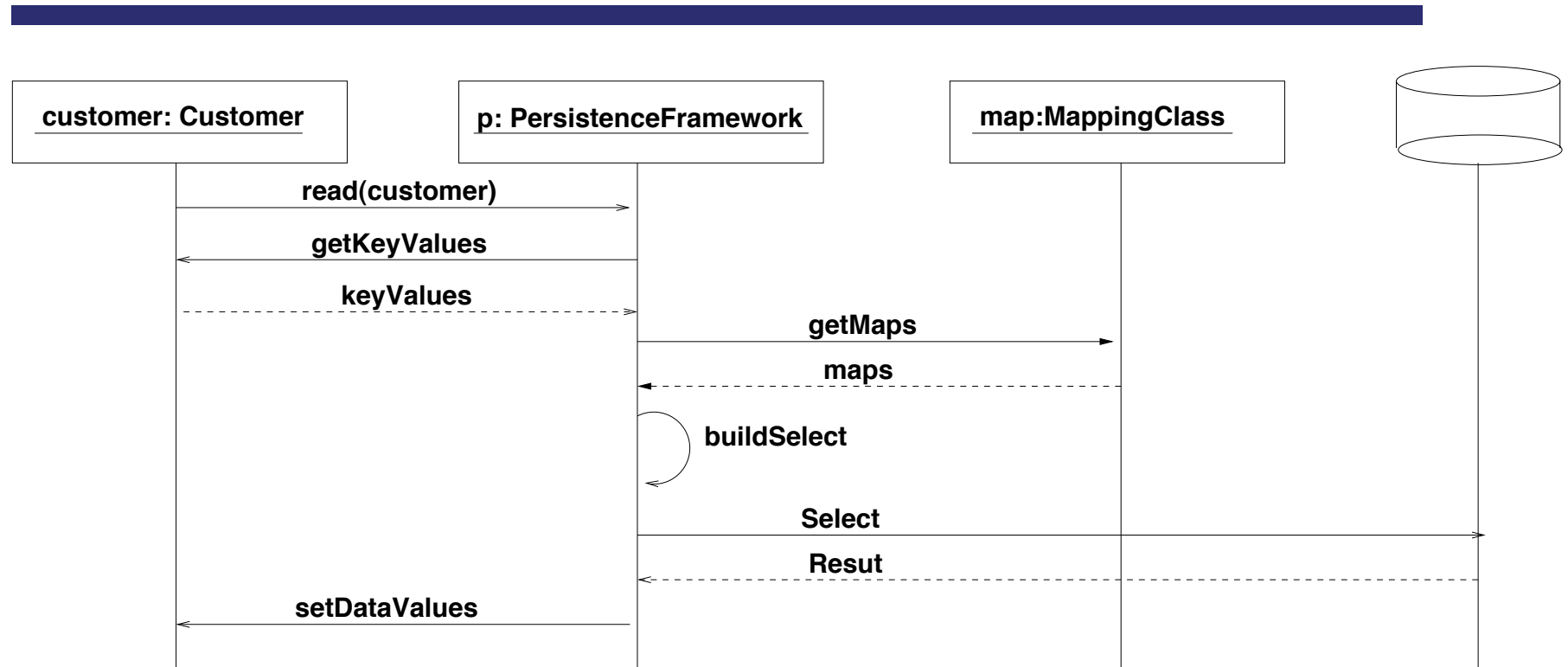
- Un **persistence framework** incapsula pienamente la logica di accesso al DB
- Dei **meta-dati** rappresentano la corrispondenza tra domain classes e tabelle, nonché le associazioni tra le domain classes

## ORM: Persistence framework (cont.)

---

- Offrono funzionalità di base (CRUD)
- altre funzionalità:
  - transazioni
  - gestione della concorrenza
  - caching

## ORM: Persistence Framework (cont.)



## Riepilogo

---

### Forza bruta

#### Vantaggi:

- semplicità
- rapidità di sviluppo
- possibilità di accedere a DB mal progettati o pre-esistenti all'applicazione

## Riepilogo (cont.)

---

### Forza bruta

#### Svantaggi:

- accoppiamento diretto tra applicazione e DB
- chi sviluppa l'applicazione deve conoscere dettagli sul DB
- difficile modifica del DB
- difficile riuso dell'applicazione

## Riepilogo (cont.)

---

### DAO

#### Vantaggi:

- incapsulamento
- possibilità di accedere a DB mal progettati o pre-esistenti all'applicazione
- facilità di riuso dell'applicazione
- minore accoppiamento fra le classi di dominio (solo dovuto alle dipendenze esplicitate dal class diagram)

## Riepilogo (cont.)

---

### DAO

#### Svantaggi:

- c'è ancora accoppiamento tra persistence classes e DB
- chi sviluppa l'applicazione (le persistence classes) deve conoscere dettagli sul DB
- può essere dipendente dalla tecnologia

## Riepilogo (cont.)

---

### Persistence Framework

#### Vantaggi:

- chi sviluppa l'applicazione non deve conoscere dettagli sul DB
- Facilità di cambiare la corrispondenza tra oggetti e DB (in seguito a scelte dovute alle prestazioni)
- facilità di riuso dell'applicazione e anche del persistence framework

## Riepilogo (cont.)

---

### Persistence Framework

#### Svantaggi:

- difficoltà di accedere a DB mal progettati
- decremento delle prestazioni (specie se il framework non è costruito con accortezza)
- può essere dipendente dalla tecnologia

## Per il progetto da realizzare

---

Si deve realizzare il progetto di un **piccolo sistema informativo**, costituito da:

- una base di dati relazionale, interrogabile mediante SQL
- una applicazione Java che si interfaccia alla base di dati attraverso JDBC

**L'approccio con Hibernate è quello richiesto se possibile, altrimenti è richiesto l'approccio DAO**

## Nota importante

---

Il problema del disallineamento esiste sia tra il dominio delle classi dell'applicazione e lo schema ER, sia tra le classi UML dell'applicazione e lo schema ER

- disallineamento sul trattamento delle generalizzazioni
- se si segue la metodologia di progettazione vista nel corso di Progettazione del Software, nella fase di progetto, sono definite delle classi Java per realizzare tipi UML, e.g. `Indirizzo`
- classi UML di tipo `TipoLink` non hanno una corrispondenza in ER