SYSC-4806 Software Engineering Laboratory

Persistence to a Database:
Object-Relational Mapping

Object-Relational Mapping (ORM)

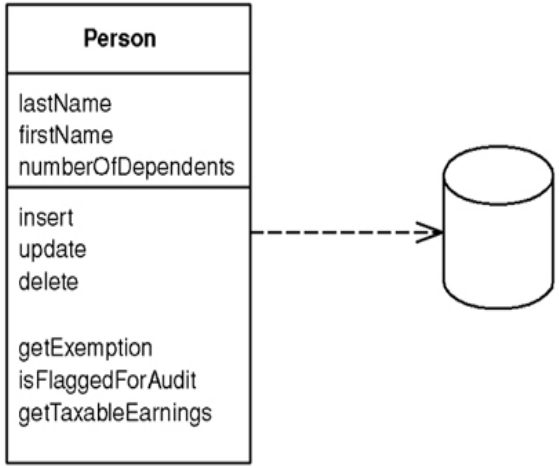
- A popular pattern best described in Martin Fowler's Patterns of Enterprise Application Architecture
  - allows OO-style use of a relational database
  - forces:
    - writing SQL in code is error-prone (no compile-checking)
    - code becomes independent of database SQL dialect
      - makes it easier to switch databases
      - makes it possible to have different DBs in different environments
    - not good practice to mix different languages (see also view templating: JSP, etc.)
    - but adds another layer with its own complexities and inefficiencies
      - for example, you may not realize that you are querying the database multiple times needlessly

The Patterns

- An ORM tool typically implements a Data Mapper and/or ActiveRecord
  - some way to abstract away the database connection
- It also helps with behavioral issues such as transactional operations on multiple dependent objects
- And it helps bridge the *impedance mismatch* between an OO and a relational representation of data

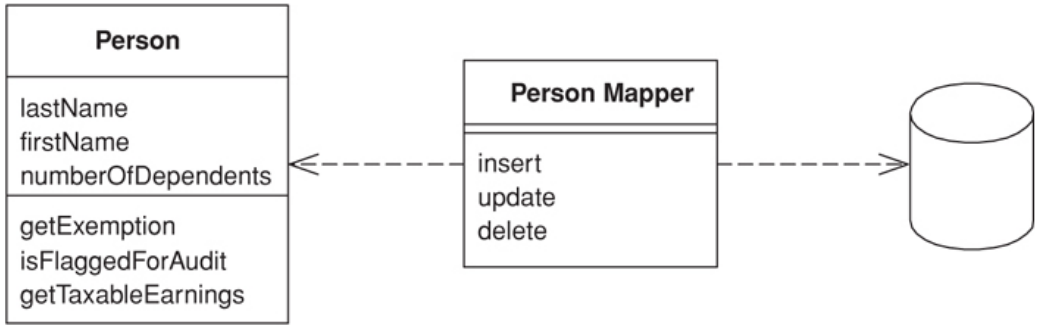
Active Record

- The simpler ORM pattern, simple enough that you can implement it yourself
- The idea:
  - one-to-one mapping of each model class to a database table
    - and a one-to-one mapping of each model instance to a row in that table
  - provide supporting CRUD methods
- Supported extensively in frameworks such as Rails or Play
  - dynamic languages (such as Ruby) can do more for you here
    - e.g., use reflection to dynamically generate those CRUD methods



Data Mapper

- The more complex pattern
- You likely will never have to implement one yourself, but rather use an ORM provider
- The idea: the mapper sits between the domain objects and the database, in a way that they stay decoupled from one another
  - this way they domain objects and the database can evolve independently from one another
  - the mapping between the domain object attributes and the table columns is specified programmatically or in a configuration file

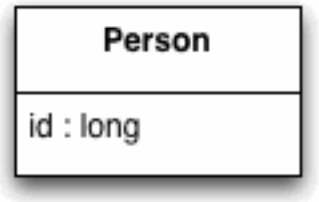


Impedance mismatch

- references vs foreign keys
- has-many relationship: collection (multi-valued attribute!) vs normalized tables with foreign keys
- many-to-many relationships: in a relational database they require an intermediate join table
- inheritance
- for these issues, Fowler has captured a number of Structural Mapping Patterns
  - Identity Field
  - Association Table Mapping
  - Single Table Inheritance

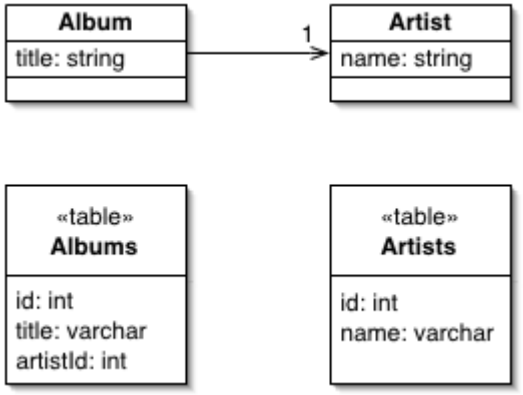
Structural Mapping Patterns: Identity Field

- In OO, you use can identify an object by its reference
- In a relational database, you identify a row by its primary key
- In ORM, specify an existing attribute as the primary key, or just add a variable for that purpose



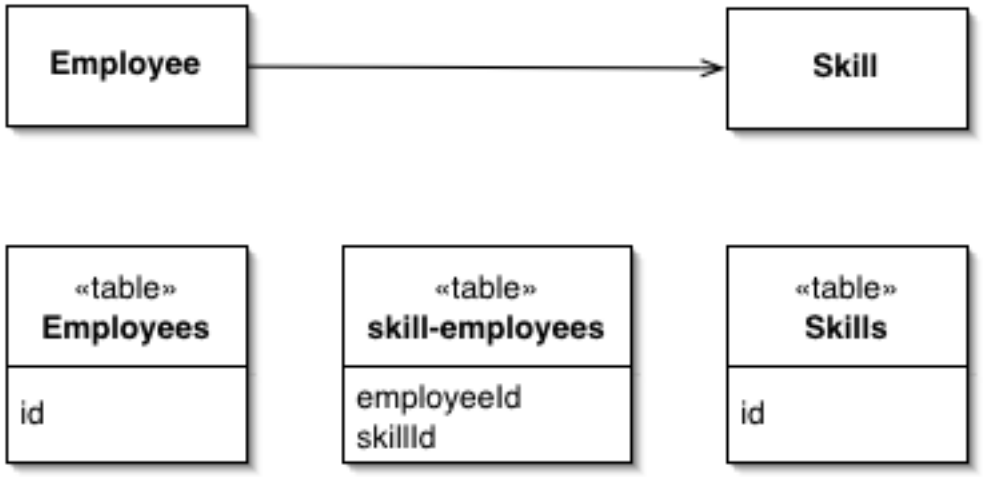
Structural Mapping Patterns: Foreign Key Mapping

- In OO, a has-many relationship is captured by a collection
- In a relational database, this corresponds to a multivalued attribute, but would not comply to the First Normal Form
  - create as many rows as values, and use a foreign key to refer to the single-valued end



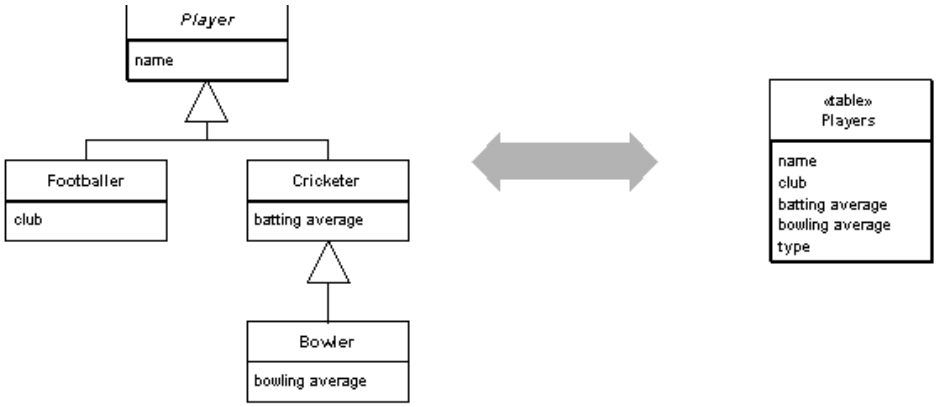
Structural Mapping Patterns: Association Table Mapping

- A many-to-many association cannot rely on a simple foreign key mapping, as there can be more than one ends
  - Use an extra table to record the relationship



Structural Mapping Patterns: Single Table Inheritance

- In OO, a subclass inherits instance variables of the superclass
- in relational databases, no inheritance. Having separate tables for the superclass and the subclass would work, but would often involve making expensive join operations
  - one possibility is to consolidate the hierarchy into a single table, and use a type column to differentiate
  - some waste of space but usually worth it.



ORM support

- ORM is supported in many languages by many libraries and frameworks:
  - Hibernate in Java, ActiveRecord in Ruby/Rails, etc.
  - in Java there is now a standard called Java Persistence API (JPA)
    - Hibernate is one of many vendors implementing JPA
    - JPA is a layer on top of JDBC, the traditional driver API for databases

JPA Howto

- To persist data to a database using JPA you need:
  - a database (duh!), e.g. SQLite, MySQL, PostgreSQL…
  - a JDBC driver for that database
  - a JPA implementation: Hibernate, Eclipse's, etc.
  - to annotate the class of the objects you want to persist
  - to write a configuration file called `persistence.xml`, specifying the above

JPA Howto 1: A database

- SQLite vs full-fledged databases

JPA Howto 2: JDBC

- JDBC is the low-level API to connect Java to a database
  - you can use it to connect to a DB, and to query it using SQL
  - each database provider will have to provide a driver that implements JDBC
  - as a result, your Java queries can be fairly database-neutral

JPA Howto 3: A JPA implementation

- JPA is a JSR (Java Specification Request) that builds on top of JDBC
- allows more seamless and transparent interaction with the DB. It is the Data Mapper.
- but it's just a standard! Need actual implementation
  - recall how SAX was just a standard for XML parsing
- Hibernate was an early ORM provider, and is one of the JPA providers
- EclipseLink, etc.

JPA Howto 4: Annotate your classes

- Full details here
- @Entity to annotate the class to be persisted
- @Id to annotate the primary key to be used
- Associations: @OnetoOne, @OneToMany, @ManyToOne, @ManyToMany
  - use them to navigate from one entity to another
  - annotate the method that allows you this navigation
  - remember from Association Table Mapping that the foreign key lives on the other end of the one-to-many relationship
- best done with an example (scroll down to Player and Team classes)

JPA Howto 5: persistence.xml

- The important elements are:
  - persistence-unit
  - provider
  - class
  - property
- best done with an example (scroll down to fig. 2)
- in newer JPA providers, this step has become unnecessary!

JPA Howto 6: querying your objects

- you first need an entity manager
  - this may be *injected* for you (more on this next week!)
- you can use this entity manager to create, read, update, delete records in the database
  - operations that write to the database must be enclosed in a *transaction*
  - for more complex queries you can use a Java variant of SQL called JPQL
  - but many ORMs can provide a more OO-friendly way of retrieving objects
  - also don't forget about cascading operations
- best done with an example