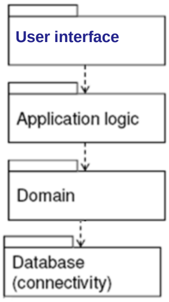
260CT
Software Engineering

Dr. Yih-Ling Hedley
Email: aa0817@coventry.ac.uk

Layered Architecture: Review



Boundary classes

Control classes

Data/entity classes

Data management classes

Data Persistence

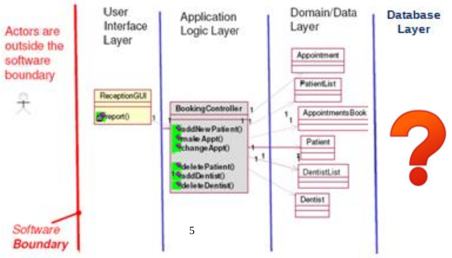
- **Persistent data or objects (**in object-oriented systems)
  - must exist from one execution of an application to another or be shared among different instances of applications
  - should exist even when the system is not active
- **Transient data or objects (**in object-oriented systems)
  - exist in memory and are discarded when an application terminates.

Persistence Mechanisms

- **Files** hold data
- **Database management systems (DBMS)**
  - **Relational DBMS**: holds tables of data
  - **Object DBMS**: holds objects (containing attributes)
  - **Object-Relational DBMS**: combines relational databases (simplicity and efficiency) and ability of object databases to store complex objects

Layered Architecture: PDS Example

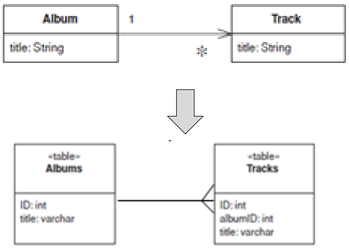
- **Layered Architecture**:
  - **Data access layer**: (also database layer, data source layer) an external data source, e.g. a database.



Mapping to Relational Database 1

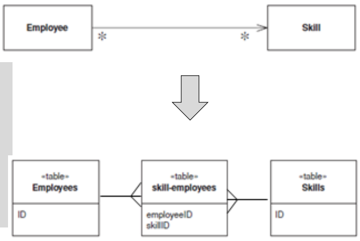
- **One-to Many Mapping of objects to database tables: Foreign Key Mapping**

Use **a foreign key** to map a field - place an unique identifier of the table on one side to the table on many side



Mapping to Relational Database 2

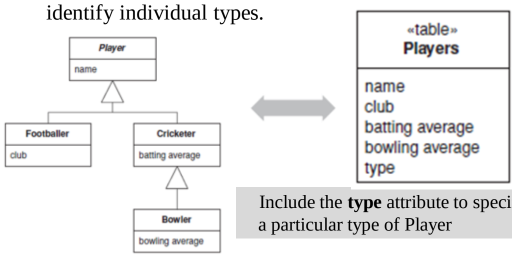
- **Many to Many Mapping of objects to database tables: Association Table Mapping**



Place unique identifiers of the tables on the **association table** created to resolve M-M association

Mapping to Relational Database 3

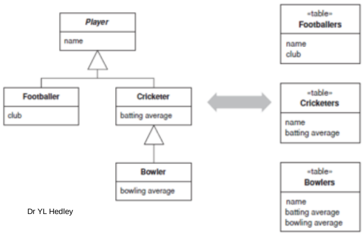
- **Inheritance Objects to Database Table Mapping**
  - Option 1: **Single Table Inheritance** - uses one table to store all the classes in a hierarchy, with the ability to identify individual types.



Include the **type** attribute to specify a particular type of Player

Mapping to Relational Database 3.1

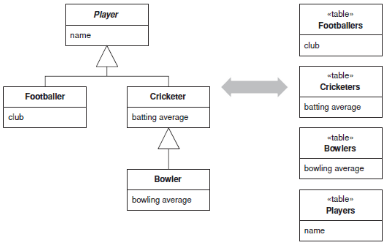
- **Inheritance Objects to Database Table Mapping**
  - Option 2: **Concrete Table Inheritance** - uses one table to store each concrete class in a hierarchy, with the sup-class identifier in each of the concrete classes



Include the **name** attribute of the super-class, Player, in each of its sub-classes

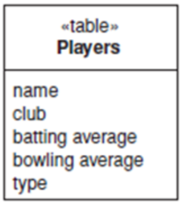
Mapping to Relational Database 3.2

- **Inheritance Objects to Database Table Mapping**
  - Option 3: **Class Table Inheritance** - uses one table for each class in a hierarchy



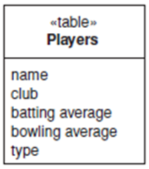
Comparison of Mapping Methods 1

- **Comparison of different mapping options: trade-offs depending on duplication of data structure and speed of access.**
  - **Single Table Inheritance:**
    - Advantage: **easier modification and avoids joins**, as all in one table



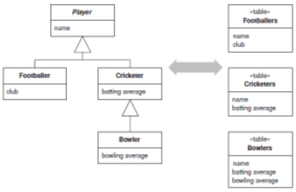
Comparison of Mapping Methods 1.1

- **Single Table Inheritance:**
  - Drawbacks:
    - ✓ **wasted space**, as each row has to have columns for all possible subtypes which leads to empty columns. However, many databases may successfully compress wasted table space.
    - ✓ **large size of table**, resulting in a bottleneck for accesses.



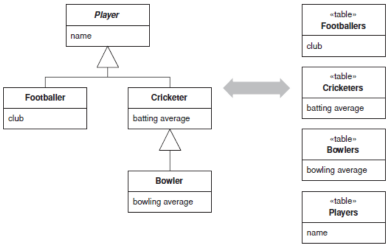
Comparison of Mapping Methods 2

- Concrete Table Inheritance:
  - **avoids the joins**, allowing a single object to be pulled from one table
  - However, **inflexible to changes**, i.e., with any change to a super-class, all the tables (and the mapping code) are required to be altered. The alteration of the hierarchy itself can cause even bigger changes.



Comparison of Mapping Methods 3

- **Class Table Inheritance:**
  - needs multiple joins to load a single object, which usually **reduces performance**.

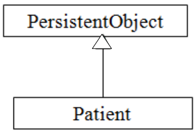


Data Management Classes: Design

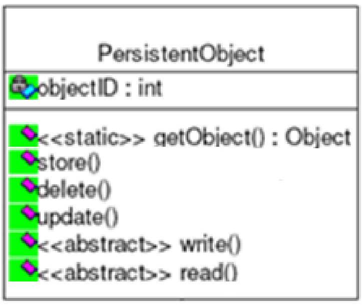
- Methods to management data:
  - **PersistentObject:** allow all persistent objects to inherit from a **PersistentObject class**
  - **Data broker classes**: database broker approach, use **data storage classe**s to manage objects

Persistent Object Approach: Direct Mapping

- **Direct Mapping**
  - The **Persistent Object** approach means that individual data/entity classes have to deal with **mapping objects** to and from the database.
  - Design an abstract super-class **Persistent Object** that encapsulates the mechanisms for an object of any class to store itself in and to retrieve itself from a database

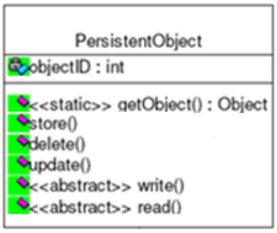


Persistent Object: Persistent Object Class



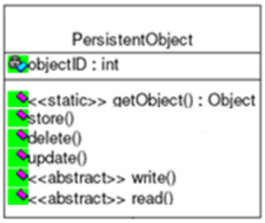
Persistent Object: Methods $_1$

- **store()**: inserts a new object into file or DB (database)
- **delete()**: deletes an existing object from file or DB
- **update()**: updates existing object in the file or DB



Persistent Object: Methods $_{2}$

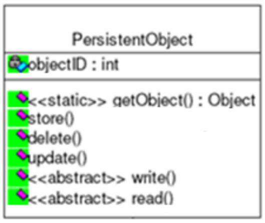
- **store(), delete() and update() need to call write() and read() to:**
  - break an object down into its attributes and write them to database
  - read the attributes back and reconstruct an object.



PersistentObject
objectID : int
<<static>> getObject() : Object
store()
delete()
update()
<<abstract>> write()
<<abstract>> read()

Persistent Object: Methods $_{3}$

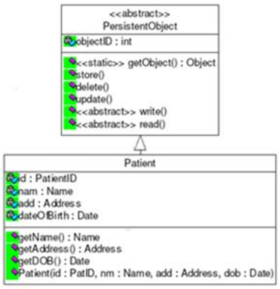
- **write()**: write to file or DB (database)
- **read()**: read from file or DB
  - Implementation: both **abstract methods declared in the super-class** (i.e. their **method bodies are not implemented in the super-class**)



PersistentObject
objectID : int
<<static>> getObject() : Object
store()
delete()
update()
<<abstract>> write()
<<abstract>> read()

Persistent Object: Methods $_{4}$

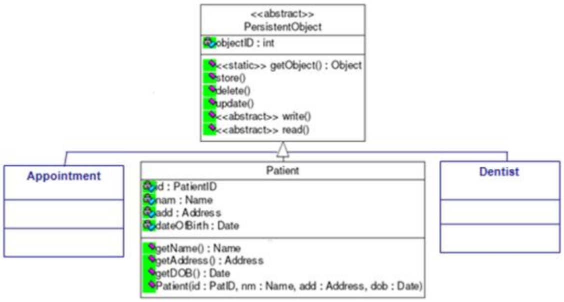
- **write() and read() methods:**
  - will be different for every persistent data class (Patient, Appointment, Dentist, etc). **So write() and read() have to be defined in each different data class** (such as Patient, Appointment and Dentist).



<<abstract>>
PersistentObject
objectID : int
<<static>> getObject() : Object
store()
delete()
update()
<<abstract>> write()
<<abstract>> read()

Patient
id : PatientID
nam : Name
add : Address
dateOfBirth : Date
getName() : Name
getAddress() : Address
getDOB() : Date
Patient(id : PatID, nm : Name, add : Address, dob : Date)

Persistent Objects



<<abstract>>
PersistentObject
objectID : int
<<static>> getObject() : Object
store()
delete()
update()
<<abstract>> write()
<<abstract>> read()

Appointment

Patient
id : PatientID
nam : Name
add : Address
dateOfBirth : Date
getName() : Name
getAddress() : Address
getDOB() : Date
Patient(id : PatID, nm : Name, add : Address, dob : Date)

Dentist

Persistent Object Approach

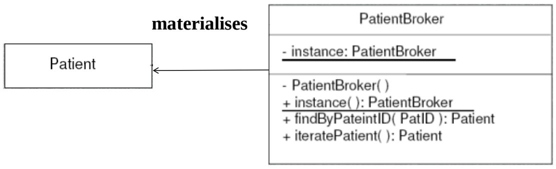
- **Direct Mapping - Persistent Object Approach**
  - **One PersistentObject class** to all classes: **very strong coupling all business classes (data classes) to a utility class**
  - data classes have **poor separation of concerns (low cohesion)** and thus it is difficult to reuse.

Data Brokers Approach: Indirect Mapping:

- **Indirect Mapping**
  - In this approach using **data brokers**, there is **good separation of concerns (higher cohesion and lower coupling)**. All the **persistence related methods/functionality are in the data broker classes**, and are transparent to the data classes.
- **Database broker framework**
  - **separates business objects from data storage classes**.
  - **An associated database broker class** is required **for each business class** to be persistent.
  - The broker class provides **mechanisms to materialise objects** form the database and dematerialise them back.

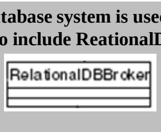
Data Broker Classes

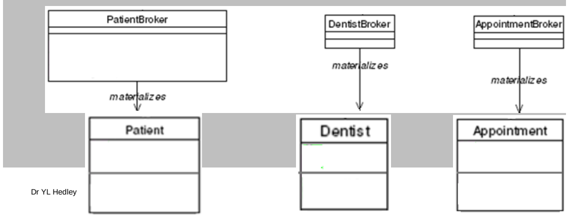
- Data Broker - Example:
  - **PatientBroker** is responsible for the storage and retrieval of **Patient** object instances


materialises
Patient
PatientBroker
- instance: PatientBroker
- PatientBroker( )
+ instance( ): PatientBroker
+ findByPateintID(PatID): Patient
+ iteratePatient( ): Patient

Data Management Classes: Exercise 1

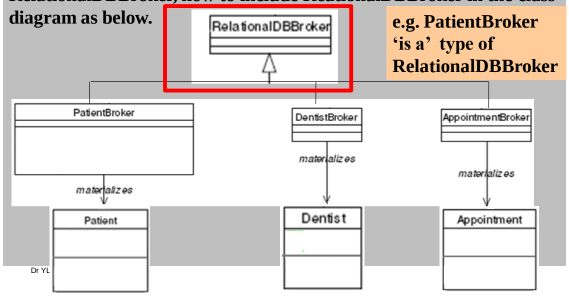
Assume that a relational database system is used with a class called RelationalDBBroker, how to include ReationalDBBroker in the class diagram as below.





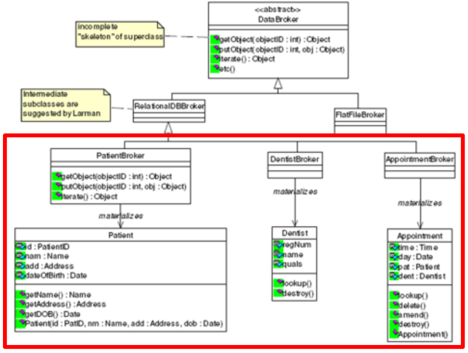
Data Management Classes:
Exercise 1 Feedback

Assume that a relational database system is used with a class called RelationalDBBroker, how to include ReationalDBBroker in the class diagram as below.



e.g. PatientBroker ‘is a’ type of RelationalDBBroker

Data Broker Classes: Example


<<abstract>>
DataBroker
getObject(objectID : int) : Object
putObject(objectID : int, obj : Object)
iterate() : Object
etc()
incomplete "skeleton" of superclass
Intermediate subclasses are suggested by Larman
RelationalDBBroker
FlatFileBroker
PatientBroker
getObject(objectID : int) : Object
putObject(objectID : int, obj : Object)
iterate() : Object
DentistBroker
AppointmentBroker
materializes
materializes
materializes
Patient
id : PatientID
nam : Name
add : Address
dateOfBirth : Date
getName() : Name
getAddress() : Address
getDOB() : Date
Patient(id : PatID, nm : Name, add : Address, dob : Date)
Dentist
regNum
name
quals
lookup()
destroy()
Appointment
time : Time
day : Date
pat : Patient
dent : Dentist
lookup()
delete()
amend()
destroy()
Appointment()