Application Program Development

Segment : ORM's

Mahboob Ali

Outcomes

- What are ORM's?
- Benefits of ORM's.
- Disadvantages of ORM's.
- ORM Patterns
- Best practices for building persistence layers

Why the need?

- Developing software is a complicated task.
- We came a long way in software development now, we passed
  - Assembler days
- We have now
  - Interpreters
  - Higher level languages
  - Tools and Techs
- With all these new technologies we still face challenges in development
- One of the Biggest challenge comes from Database.

Why the need?

- Enterprise level of application usually involve persistent data.
- The data is persistent because it needs to be around between multiple runs of the program.
- There's usually a lot of data—a moderate system will have over 1 GB of data organized in tens of millions of records.
- Usually, many people access data concurrently. For many systems this may be less than a hundred people but for other it can be over thousands.
- With so much data, there's usually a lot of user interface screens to handle it. It's not unusual to have hundreds of distinct screens.

Different types of systems.

- Consider a B2C (business to customer) online retailer: People browse and—with luck and a shopping cart—buy. For such a system we need to be able to handle a very high volume of users.
- Contrast this with a system that automates the processing of leasing agreements. In some ways this is a much simpler system than the B2C retailer's because there are many fewer users—no more than a hundred or so at one time. Where it's more complicated is in the business logic. Calculating monthly bills on a lease, handling events such as early returns and late payments, and validating data as a lease is booked are all complicated tasks, since much of the leasing industry's competition comes in the form of little variations over deals done in the past.
- A third example point is a simple expense-tracking system for a small company. The only data source is a few tables in a database. As simple as it is, a system like this is not devoid of a challenge. You have to build it very quickly and you have to bear in mind that it may grow as people want to calculate reimbursement checks, feed them into the payroll system, understand tax implications, provide reports for the CFO.

What are patterns?

- Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

- A key part of patterns is that they’re rooted in practice. You find patterns by looking at what people do, observing things that work, and then looking for the “core of the solution.” It isn’t an easy process, but once you’ve found some good patterns they become a valuable thing.

The Three Principal Layers/ 3-Tier Architecture

- Presentation
  - **Presentation** logic is about how to handle the interaction between the user and the software. This can be as simple as a command-line or text-based menu system, but these days it’s more likely to be a rich-client graphics UI or an HTML-based browser UI.
- Data Source
  - **Data source** logic is about communicating with other systems that carry out tasks on behalf of the application. These can be transaction monitors, other applications, messaging systems, and so forth.
- Domain Logic
  - The remaining piece is the domain logic, also referred to as business logic. This is the work that this application needs to do for the domain you’re working with. It involves calculations based on inputs and stored data, validation of any data that comes in from the presentation and figuring out exactly what data source logic to dispatch, depending on commands received from the presentation.

Object Oriented

- If you've decided to use Object Oriented Language to write your application in, then what will be the next steps?
  - Process to gather requirements
  - Creating a design (UI/ UX)
  - Development (coding)
  - Testing
- During the design phase what is your approach to access the data?
  - Are you going to use stored-procedures for CURD?
  - Are you going to create a custom data access layer using ADO.NET and COM+?
  - Or may be get a widget to do all this?

Trouble with Objects

- From an OO standpoint, the problem with these persistence mechanisms is that their core abstractions are not objects

- They are tables with rows and columns (RDBMS)

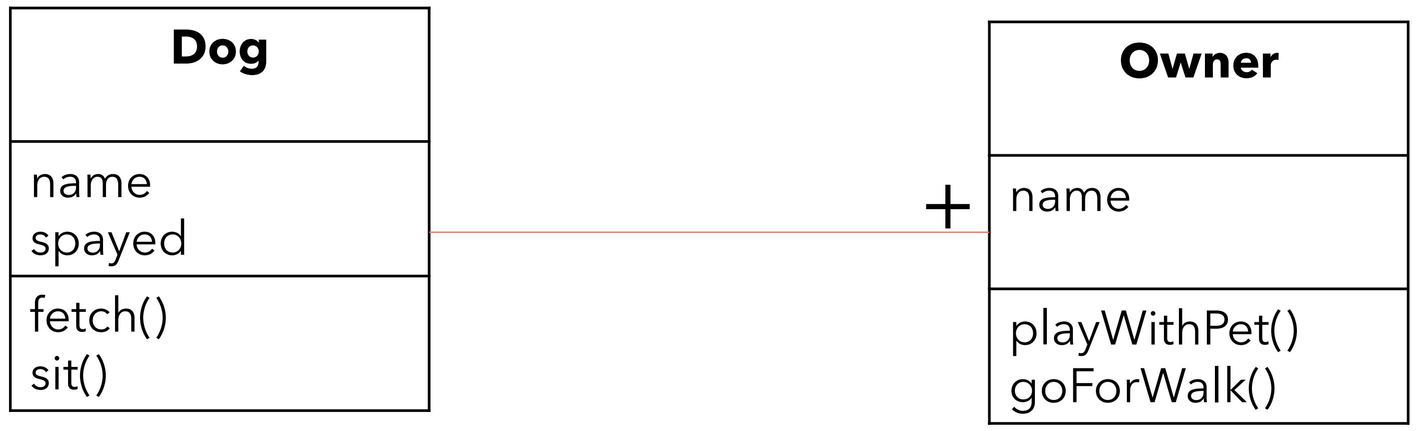
Or

- They are (some variation on) key-value pairs (NoSQL)

Problems Conti…

- The OO world, on the other hand, has
  - Classes, sub-classes, inheritance, associations
  - Objects, atts, methods, polymorphism
- These concepts do not easily map into the abstractions of persistence mechanisms
  - Even the creation of serialization mechanisms is non-trivial with the work that has to go in to traversing and reconstituting an object graph

An Example


Dog
name
spayed
fetch()
sit()
+
Owner
name
playWithPet()
goForWalk()

Discussion

- Think about how you would represent the previous UML diagram in a relational database
  - In the system, you will have Dog objects and Owner objects and some of them will be related to each other

- You will at least have
  - a table called dogs to store Dog instances and
  - a table called owners to store Owner instances

- Indeed, this is a convention of many object-relational mapping systems
  - class names are singular; table names are the associated plural form of the word: Person ⇒ People ; Cat ⇒Cats; etc.

More Discussion

- Furthermore, for each table
  - you would have columns that correspond to each attribute (plus an implicit id column)
  - each row would correspond to an instance of the class
    - a spayed dog named Fido might have a row like:
    - 1 | Fido | true

How do we handle the relationship between Dog and Owner?

- Based on the diagram
  - Each owner has a single dog
  - Each dog has at least one owner

- This means that two owners can own the same dog
  - Owner participates in a “has_one” relationship with Dog
  - Dog participates in a “has_many” relationship with Owner

How do we handle the relationship between Dog and Owner?

- The short answer is
  - foreign key relationships and join tables
- The somewhat longer answer is that most object-relational mapping systems have ways to specify these relationships
  - They then take care of the details automatically
  - You might see code like:
    - List<Owner> owners = dog.getOwners();
- Behind the scenes, the method will hide the database calls required to find which owners are associated with the given dog

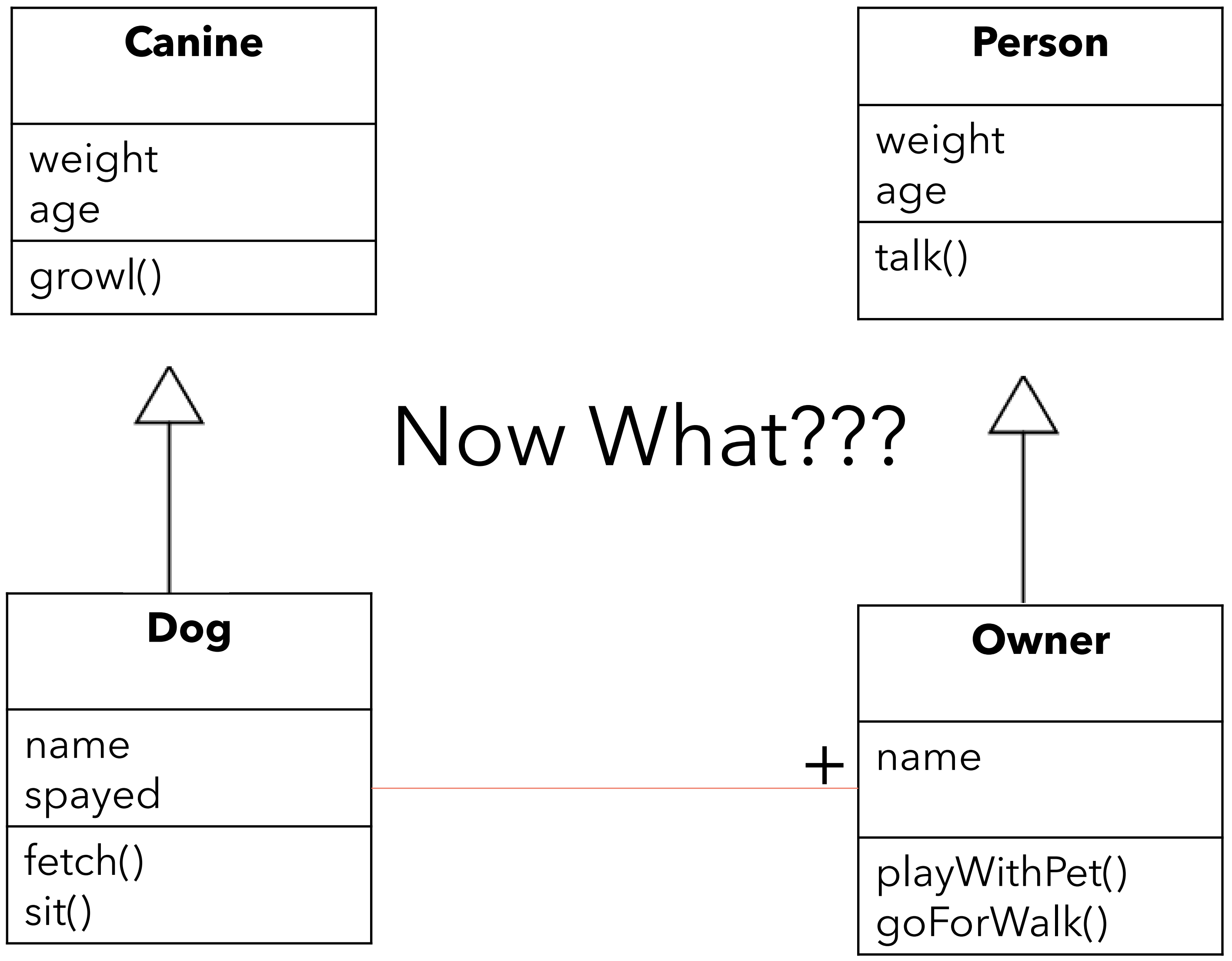
How do we handle the relationship between Dog and Owner?

- The relationship between Dog and Owner can be handled such that
  - Each instance of dog is assigned a unique id
    - 1 | Fido | true
    - 2 | Spot | false
  - Likewise owners
    - 1 | Ken
    - 2 | Max
- A third table is then used to maintain mappings between them
    - 1 | 1 ; 1 | 1 ; 2 | 2 ; 2
- This says that Fido is owned by Ken and Max and Spot is owned by Max

Some more Discussion

- That third table is known as a join table and has the structure
  - dog_fk | owner_fk
  - "1 | 1" in a row says that dog 1 is owned by owner 1
- When it is time to implement the code
  - List<Owner> owners = dog.getOwners();
- Then
  - the code gets the id of the current dog
  - asks for all rows in the join table where dog_fk == "id of current dog"
- this provides it with some number of rows; each row provides a corresponding owner_id which is used to lookup the names of the associated owners

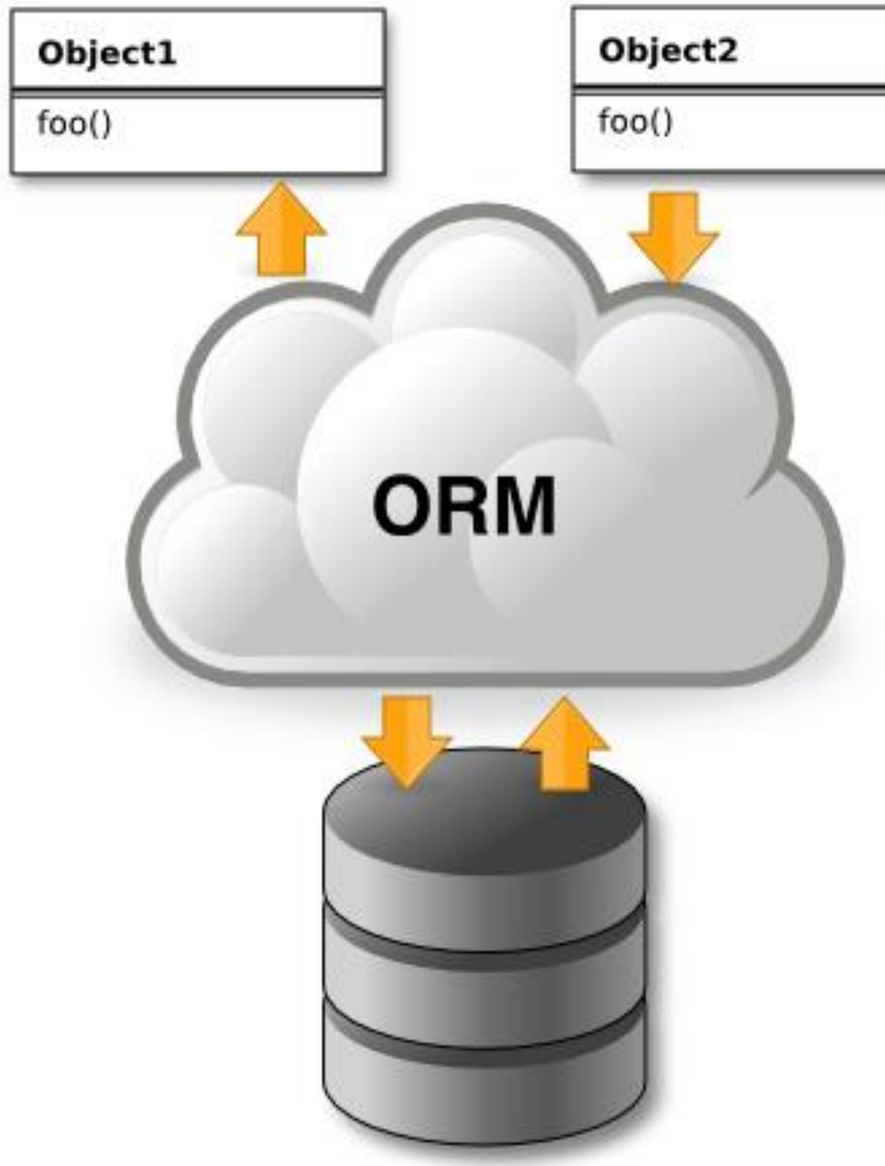
Complication



What is ORM then?

- Object-relational mapping is a programming technique
  - for converting data between incompatible type systems
  - by creating objects using OO programming languages
  - essentially wrapping tables or stored procedures in classes and interact with database using their methods and properties
- ORM is an automated way of
  - connecting an object model, sometimes referred to as a domain model, to a relational database by using metadata as the descriptor of the object and data.
- This creates, in effect, a "virtual object database

- ORM is the act of
  - connecting object code, whether it is in C#, Java, or any other object-oriented language, to a relational database.
- This act of mapping is an efficient way to overcome the mismatch that exists between object-oriented development languages and relational databases.
  - Such a mismatch can be classified as an inequality between the native object-oriented language operations and functions and those of a relational database



Introduction to ORM

Relational Models vs. OO Models

- Types (ex. char(n) versus string)
- Identity (keys versus address or equality)
- Relations (foreign keys versus references)
- Many to many relations require linking table in SQL
- Inheritance and polymorphism

Benefits of ORM

- Automates the Object-to-table and Table-to-object conversions.
- work in the OO model
  - without having to worry about the underlying data structure
- Allow us to easily save objects to the database and load them from the database
- Typically provide support for the full set of CRUD (create, read, update, delete) operations
- Save time and money (time to market)
- Focus on the business logic
  - rather than database/persistence logic

Selecting a suitable ORM

- Object-to-database mapping
  - The single most important aspect of an ORM tool.
  - Ability to map business objects to back-end database tables.
- Object cashing
  - Ability to enable object/ data cashing to improve performance.
- GUI mapping
  - Can survive without it but it will be bonus to have in an ORM tool.

Selecting a suitable ORM

- Dynamic Querying
  - Another important aspect in an ORM tool, which allows the different projection and classless queries based on the user input.
  - LINQ-to-SQL and EF both support this.
- Lazy loading
  - Improves the performance by optimizing the memory utilization of database server.
  - This is done by prioritizing the components, should they be loaded on first come first bases or pre-loaded as the program starts.
- Nonintrusive persistence
  - This is an important one, meaning no more class, interface or inheritance based specific relations.