Service Provider Interface & Data Access Layers

Technion – Institute of Technology
236700
What are Frameworks?

- In our context, frameworks are a generic, reusable solution that can be adapted to many different needs.
- Unlike libraries, frameworks have inversion of control and are modifiable:
  - IOC: Frameworks invoke our code, not the other way around.
  - Extendible: We can add specialized code and case handling.
- Modern programming isn't so much about programming as it is about gluing together existing frameworks:
  - Virtually any infrastructure you want to do has been done before:
    - The API is battle-tested, and there are (usually) a lot less bugs.
    - You only have to fill the blanks with your business logic.
- We've already seen some "small" framework examples:
  - JUnit, Mockito, Guice.
- But some frameworks are much larger in scope.
Three Tier Architectures

- Most modern applications use a three-tier system
- This is especially common for web applications which will our focus in the coming weeks

- **UI**
  - Interacts with the user
  - Translates user actions to commands

- **Business Logic**
  - Processes commands
  - Makes logical decisions

- **Data**
  - Persists and indexes data (our focus today)
One API to rule them all

The most common data solution is relational databases

• Good: SQL is a the de-facto interface with relations databases
• Bad: There are a lot of different providers / engines
  • And new ones pop up occasionally
• We don't want to be tied down to a single implementation
  • Different solutions have different pros and cons
  • Some engines are good for development, some for testing, and others for production
• But defining a uniform API and implementing it ourselves for every provider is time consuming and error-prone
• Solution: make the providers implement the uniform API!
  • This can usually only be done by the standard library
Service Provider Interface (SPI)

- We would like to provide users with a uniform API
- Enable various vendors to supply their own implementation for the interfaces defined by the API
- In our example – Data Access Layer (DAL)

![Diagram of Service Provider Interface (SPI)]
Evaluating SPIs

• Goals:
  • Keep everybody (vendors/clients) happy, or at least equally disappointed
  • As always – ‘good’ API for clients
    • If it’s a bad API, no one would use it
  • But also – easy to implement Service Providers to vendors
    • If it’s hard to implement, no one would supply it
    • We can usually require a small abstract subset and implement a more complete API on top of that

• Nice to have:
  • Enabling all kinds of vendors to hook in
    • E.g. is it possible for a non-relational DB vendor to hook to this DAL?
  • Testability
    • Enabling mock or fake implementations
Implementing a DAL SPI

• **Providers** only need to implement abstract classes and interfaces in the API
  • In other words, we can provide default implementation
  • That way we can provide the clients with **Rich API**
• **Users** of the DAL will be able to transparently use the new vendor’s implementation
  • (E.g. switch between MySQL/DB2/SQLite DB engines without changes* to their code)
  • *This is not entirely true as SQL has dialect differences
Java Database Connectivity (JDBC)

- An SPI developed by Sun Microsystems, as part of JDK 1.1 (1997)
- Supply unified database access
- Strives to make the underlying SQL-DB platform transparent (Implementation-wise)
  - As much as possible
JDBC

• Select vendor:

```java
Class.forName("com.mysql.jdbc.Driver");
Connection connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/mydb", ...);
```

• Using the API:

```java
Statement stmt = connection.createStatement();
ResultSet res = stmt.executeQuery("SELECT * FROM ...");
```

• Hidden assumptions:
  • `ResultSet` is an iterator on lines of results
  • Results are retrieved by stating the column name
    ```java
    String name = res.getString("Name");
    ```
  • Implicitly forces to use only `relational DB`
SQL Injections

“Can add to an API but can never remove”.

• Example:

```java
String userName = "'anything' OR X=X";
``` 

• What happens when:

```java
PreparedStatement pstmt = conn.prepareStatement(
  "SELECT Personal_data FROM where name=?");
pstmt.setString(1,userName);
ResultSet res = pstmt.executeQuery();
``` 

• Late solution:

```java
PreparedStatement pstmt = conn.prepareStatement(
  "SELECT Personal_data FROM users_table WHERE name=" + userName);
``` 

• But now all implementation have to provide implementations for the broken API as well
Summary

• JDBC was a good start but:
  • Developers basically needs to maintain two data models: DB Tables + Code objects
  • A lot of “translation” code is needed:
    • Objects to relational rows
    • Relational rows to objects
  • Abstraction level is very shallow
    • Complicated error-path
    • Queries may vary between DB implementations
  • API induces complicated code
    • Hard to keep it DRY
ORM Considerations - Composition

Inheritance/Composition

- How do we maintain object composition in Databases?

```java
class PersonalDetails {
    String id_number;
    String name
    String address;
}
class DegreeDetails {
    double average;
    String faculty;
    String degreeType;
}
class Student {
    PersonalDetails pd;
    DegreeDetails dd;
}
```

- Note that composition can go much deeper than
- Inheritance can be modeled as composition (referencing the super type)
Composition (Cont.)

Option 1: flatten everything and put it in a single table

<table>
<thead>
<tr>
<th>id_number</th>
<th>name</th>
<th>address</th>
<th>average</th>
<th>faculty</th>
<th>degree</th>
</tr>
</thead>
<tbody>
<tr>
<td>123456789</td>
<td>joe</td>
<td>foo</td>
<td>90</td>
<td>CS</td>
<td>CS_3_y</td>
</tr>
<tr>
<td>234567890</td>
<td>jim</td>
<td>bar</td>
<td>95</td>
<td>EE</td>
<td>EE_Phys</td>
</tr>
</tbody>
</table>

- **Pros:**
  - Simple queries (no joins)
  - Easy to handle entire table (e.g., drop, clear)
  - Cache locality when we want to pull everything

- **Cons:**
  - Doesn't scale, have to migrate entire tables when adding new data
  - Doesn't compose
    - Can't reuse PersonalDetails for employees
    - Employee can't have an Employee manager field
  - Slow when we want to pull only partial data, e.g., just the IDs
Composition (cont.)

• Option 2: A table for each type, and keep a **foreign key** to another table's ID

<table>
<thead>
<tr>
<th>PersonalDetails</th>
<th>DegreeDetails</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>id_number</td>
</tr>
<tr>
<td>0</td>
<td>123456789</td>
</tr>
<tr>
<td>1</td>
<td>234567890</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Students</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

Pros and cons are the inverse of the previous solution, in particular
• Composable, but...
• Have to write ugly joins when creating complex objects
**ORM – Collections**

- **“Linked List”:**

<table>
<thead>
<tr>
<th>Primary key</th>
<th>Value primary key</th>
<th>Next node key</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1239</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>4918</td>
<td>2</td>
</tr>
</tbody>
</table>

- **“Sets”:**

<table>
<thead>
<tr>
<th>Primary collection key</th>
<th>value primary key</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1239</td>
</tr>
<tr>
<td>0</td>
<td>4918</td>
</tr>
</tbody>
</table>

- **“Arrays”:**

<table>
<thead>
<tr>
<th>Primary collection key</th>
<th>value primary key</th>
<th>Array index</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1239</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>4918</td>
<td>1</td>
</tr>
</tbody>
</table>
Manual ORM – It’s a pain!

• No matter which solution we use for what problem, we have to write a lot of SQL and object-oriented code
• We're writing extremely imperative code
• We have to manually maintain, update, and migrate the database for each domain change, as well as the DAL
• Complex objects are incredibly hard to maintain
• Not DRY: each field appears in our business objects, in our SQL schemas, and in our SQL code
JDBC – demo

Example
- Students / Courses
- Registration of students to courses

Take aways
- Use `try-with-resources` block *(Using in C#)*
  - Note resources are actually final
  - ResultSet is a resource! *(As well as Connection and Statement)*
- Still lots of boilerplate / use patterns
  - Preparing statements
  - Deciphering the results
  - Handling exceptions
- Be careful of different SQL dialects
- Joining is hard
ORM Frameworks

- ORM Frameworks will:
  - Manage the DB structure
  - Map Classes to Tables
  - Map Objects to Rows
  - Maintain at most one object in memory per DB row

- We will:
  - Not worry about composition and inheritance
  - Not worry about Entity Equality
  - Write less code
  - Write simpler code
  - Write more **declarative** code
Hibernate

• Provides a declarative way of mapping objects to their corresponding data
• Defines an **external DSL** named HQL,
  • Similar to SQL, only it deals with objects
  • Instead of querying tables for rows, we query Classes for instances
  • For example, getting all students born after 1985:

```sql
FROM Student WHERE dob > '31/12/1985'
```

**Student** is a Java class

**dob** is a class member of type **Date**

Author: Gal Lalouche - Technion 2017 ©
Hibernate

- Simple configuration via Annotations
- Specification via XML is also supported
- Java Beans defines tables
  - Can also explicitly define (table/column names)
- Subclasses objects extends the tables
- Complex objects relations
  - Objects which points to other objects
  - Automatically creates and maintains relational tables
- No need to worry about SQL dialects
- As powerful as JDBC
  - We can still get down and dirty and SQL queries when we need to
Hibernate – Entity Equality

- Mapped objects now need to declare an ID member
- Hibernate resolves row-object identity

```java
@Entity
@Table(name = "Books")
public class Book {
    @Id private int id;
    private String author;

    public int getId() {...}
    private void setId(int id) {...}

    public String getAuthor() {...}
    // setters can be private for increase immutability
    // it's possible to use final fields too if you want
    private void setAuthor(String name) {...}

    ...
}
```
Hibernate – Creating sessions

• All actions need a Hibernate Session object

```java
SessionFactory factory =
    new AnnotationConfiguration().configure().buildSessionFactory();
Session s = factory.getCurrentSession();
s.beginTransaction();
// do something mutating, e.g., save new data
Book b = new Book(...);
session.save(book); // Book knows to know to save itself!
s.getTransaction().commit();
```

• Querying:

```java
List<Book> hemingwayBooks = session
    .createQuery("from Book where author = ?")
    .setString(0, "Hemingway")
    .list();
```

• Other frameworks give us better Typesafety by using fluent APIs
Hibernate - Takeaways

Takeaways:

• No need to manage tables
• No need to manually manage object relations and compositions
• `getAllCourses()` is down from 43 LOC to 8 LOC

Frameworks to the rescue!

• There is a **common problem** (ORM)
• There is a **generic solution** that involves very little code, and a bit of wiring
• Our code is much more **declarative** as a result