Software Engineering III

Patterns in Design

Ref: *Software Engineering: A Practitioner's Approach*

Roger S. Pressman

Design Patterns

- Each of us has encountered a design problem and silently thought: *I wonder if anyone has developed a solution to this?*
  - What if there was a standard way of describing a problem (so you could look it up), and an organized method for representing the solution to the problem?
- *Design patterns* are a codified method for describing problems and their solution allows the software engineering community to capture design knowledge in a way that enables it to be reused.

3

Design Patterns

- *Each pattern describes a problem that 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 the solution a million times over without ever doing it the same way twice.*
- "a three-part rule which expresses a relation between a certain context, a problem, and a solution."

4

Basic Concepts

- *Context* allows the reader to understand the environment in which the problem resides and what solution might be appropriate within that environment.
- A set of requirements, including limitations and constraints, acts as a *system of forces* that influences how
  - the problem can be interpreted within its context and
  - how the solution can be effectively applied.

5

Kinds of Patterns

- *Architectural patterns* describe broad-based design problems that are solved using a structural approach.
- *Data patterns* describe recurring data-oriented problems and the data modeling solutions that can be used to solve them.
- *Component patterns* (also referred to as *design patterns*) address problems associated with the development of subsystems and components, the manner in which they communicate with one another, and their placement within a larger architecture
- *WebApp patterns* address a problem set that is encountered when building WebApps and often incorporates many of the other patterns categories just mentioned.

6

Kinds of Patterns

- ***Creational patterns*** focus on the "creation, composition, and representation of objects, e.g.,
  - **Abstract factory pattern:** centralize decision of what factory to instantiate
  - **Factory method pattern:** centralize creation of an object of a specific type choosing one of several implementations
- ***Structural patterns*** focus on problems and solutions associated with how classes and objects are organized and integrated to build a larger structure, e.g.,
  - **Adapter pattern:** 'adapts' one interface for a class into one that a client expects
  - **Aggregate pattern:** a version of the Composite pattern with methods for aggregation of children
- ***Behavioral patterns*** address problems associated with the assignment of responsibility between objects and the manner in which communication is effected between objects, e.g.,
  - **Chain of responsibility pattern:** Command objects are handled or passed on to other objects by logic-containing processing objects
  - **Command pattern:** Command objects encapsulate an action and its parameters

7

Frameworks

- Patterns themselves may not be sufficient to develop a complete design.
  - In some cases it may be necessary to provide an implementation-specific skeletal infrastructure, called a *framework,* for design work.
  - That is, you can select a "*reusable mini-architecture* that provides the generic structure and behavior for a family of software abstractions, along with a context … which specifies their collaboration and use within a given domain."
- A framework is not an architectural pattern, but rather a skeleton with a collection of "plug points" (also called *hooks* and *slots*) that enable it to be adapted to a specific problem domain.
  - The plug points enable you to integrate problem specific classes or functionality within the skeleton.

8

Describing a Pattern

- ***Pattern name***—describes the essence of the pattern in a short but expressive name
- ***Problem***—describes the problem that the pattern addresses
- ***Motivation***—provides an example of the problem
- ***Context***—describes the environment in which the problem resides including application domain
- ***Forces***—lists the system of forces that affect the manner in which the problem must be solved; includes a discussion of limitation and constraints that must be considered
- ***Solution***—provides a detailed description of the solution proposed for the problem
- ***Intent***—describes the pattern and what it does
- ***Collaborations***—describes how other patterns contribute to the solution
- ***Consequences***—describes the potential trade-offs that must be considered when the pattern is implemented and the consequences of using the pattern
- ***Implementation***—identifies special issues that should be considered when implementing the pattern
- ***Known uses***—provides examples of actual uses of the design pattern in real applications
- ***Related patterns***—cross-references related design patterns

Thinking in Patterns

9

- Shalloway and Trott suggest the following approach that enables a designer to think in patterns:
  - 1. Be sure you understand the big picture—the context in which the software to be built resides. The requirements model should communicate this to you.
  - 2. Examining the big picture, extract the patterns that are present at that level of abstraction.
  - 3. Begin your design with 'big picture' patterns that establish a context or skeleton for further design work.
  - 4. "Work inward from the context" looking for patterns at lower levels of abstraction that contribute to the design solution.
  - 5. Repeat steps 1 to 4 until the complete design is fleshed out.
  - 6. Refine the design by adapting each pattern to the specifics of the software you're trying to build.

Design Tasks

10

- Examine the requirements model and develop a problem hierarchy.
- Beginning with a broad problem, determine whether one or more architectural patterns are available for it.
- Using the collaborations provided for the architectural pattern, examine subsystem or component level problems and search for appropriate patterns to address them.
- Repeat until all broad problems have been addressed.

Design Tasks

11

- If user interface design problems have been isolated (this is almost always the case), search the many user interface design pattern repositories for appropriate patterns.
- Regardless of its level of abstraction, if a patterns repository or individual pattern shows promise, compare the problem to be solved against the existing pattern(s) presented.
- Be certain to refine the design as it is derived from patterns using design quality criteria as a guide.

12

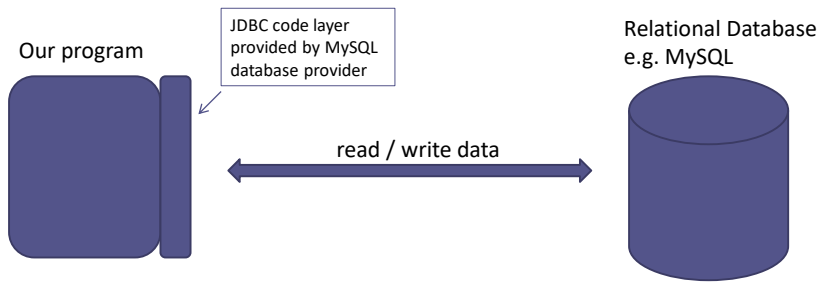
Common Design Mistakes

- Not enough time has been spent to understand the underlying problem, its context and forces, and as a consequence, you select a pattern that looks right, but is **inappropriate** for the solution required.
- Once the wrong pattern is selected, you refuse to see your error and **force fit the pattern**.
- In other cases, the problem has forces that are not considered by the pattern you've chosen, resulting in a **poor or erroneous fit**.
- Sometimes a pattern is **applied too literally** and the required adaptations for your problem space are not implemented.

The DAO Pattern

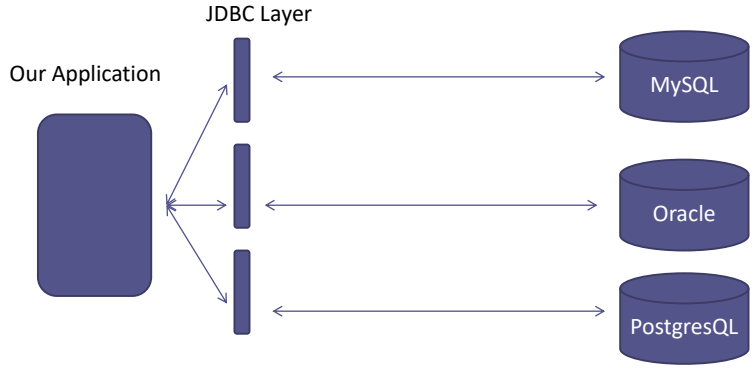
Recall - Database Connectivity

Java Database Connectivity (JDBC)



Most major RDBMS providers supply JDBC code libraries to allow java developers to easily connect to their database.

Database Connectivity
Java Database Connectivity (JDBC)



- Each RDBMS has its own implementation of the JDBC Java Interfaces
- **E.g.** Each JDBC library must have a class that implements the `Statement` interface
- `Statement` - The object used for executing a static SQL statement and returning the results it produces

Database Connectivity
Java Database Connectivity (JDBC)

- JDBC code libraries referred to as *JDBC Drivers / Connectors*
- Essentially consist of a single jar file containing all the necessary java classes
  - E.g. mysql-connector-java-x.x.xx.jar
- Once the jar file is included in the classpath, the specific JDBC classes can be used by your own code

Database Connectivity

Java Database Connectivity (JDBC)

```
// Load the database driver
Class.forName( "com.mysql.jdbc.Driver" ) ;

// Get a connection to the database
Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/test" ) ;

// Get a statement from the connection
Statement stmt = conn.createStatement() ;

// Execute the query
ResultSet rs = stmt.executeQuery( "SELECT * FROM customer" ) ;

// Loop through the result set
while( rs.next() ){
    System.out.println( rs.getString(1)+ " " + rs.getString(2) + " " + rs.getString(3));
}

// Close the result set, statement and the connection
rs.close() ;
stmt.close() ;
conn.close() ;
```

The java class from MySQL that implements the JDBC interfaces. This registers the JDBC driver with the java DriverManager

The url giving the location of the database

The SQL query to execute against the database

The ResultSet object contains the data returned from the database

Managing Data

If we have a table in a database called **customer** with the following columns

- custNumber (integer)
- custName (varchar)
- contactLastName (varchar)
- contactFirstName (varchar)
- phone (varchar)
- creditLimit (double)

Managing Data

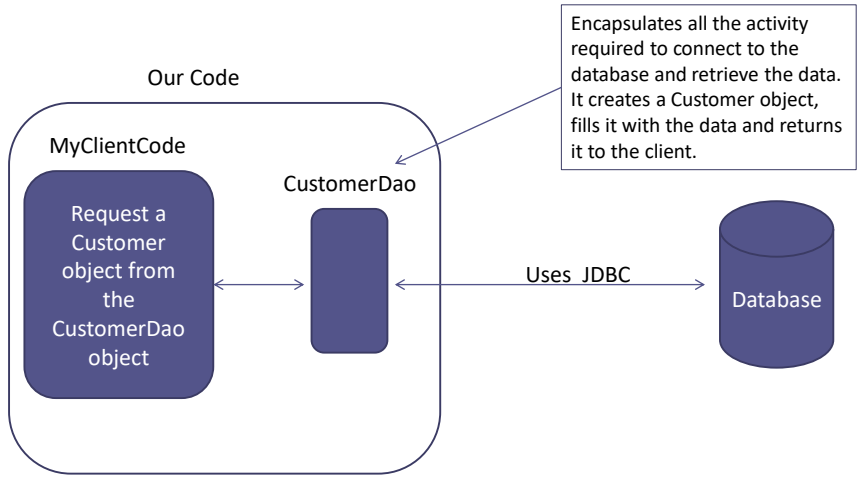
```
public class Customer {

    private int customerNumber;
    private String customerName;
    private String contactLastName;
    private String contactFirstName;
    private String phone;
    private double creditLimit;


    public int getCustomerNumber() {
        return customerNumber;
    }
    public void setCustomerNumber(int customerNumber) {
        this.customerNumber = customerNumber;
    }
        …
        …
    public void setCreditLimit(double creditLimit) {
        this.creditLimit = creditLimit;
    }

}
```

We can write a class that encapsulates the data elements that exist in the database table

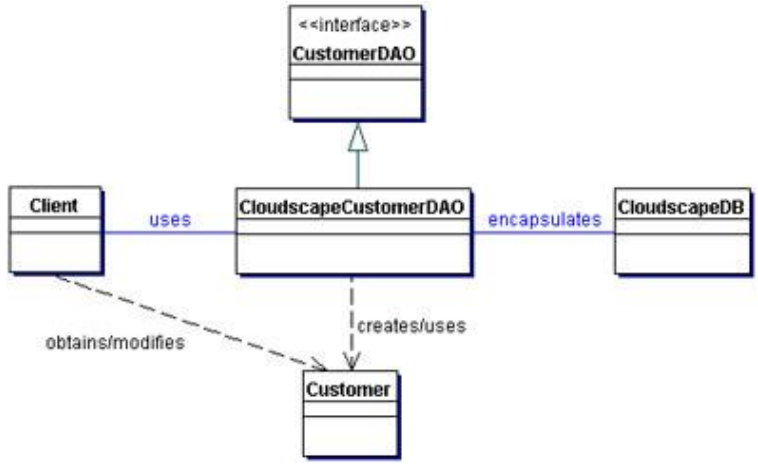
This gives us a mapping from our code to the database – an object to relational database mapping (ORM)

We can call instances of this class – ***transfer objects*** because we can use them to carry data about customers around our code.

The Data Access Object (DAO) Pattern



The Data Access Object (DAO) Pattern



Example DAO code

```
public class CloudscapeCustomerDao implements
    CustomerDao {

  public CloudscapeCustomerDao() {
    // initialization
  }

  public Vector getCustomers(...) {
    // implement search customers here using the
    // supplied criteria.
    // Return a Vector.
  }

  public Customer findCustomer(...) {
    // Implement find a customer here using supplied
    // argument values as search criteria
    // Return a Customer (Transfer Object) if found,
    // return null on error or if not found
  }

  public boolean updateCustomer(...) {
    // implement update record here using data
    // from the customer Transfer Object
    // Return true on success, false on failure or
    // error
  }
  ...
}
```

This example illustrates the methods that our DAO class might have

Note, we still need to add database connection handling code

Example client code using a DAO

```
    …
// Create a DAO
    CustomerDAO custDAO = new CloudscapeCustomerDAO();
    …


// Find a customer object. Get the Transfer Object.
    Customer cust = custDAO.findCustomer(...);
    …
```

JDBC – PreparedStatement

- We have used the `Statement` class to send SQL to the DBMS.
- SQL is sent to the DBMS which has to compile it before execution.
- `PreparedStatement` objects are given the SQL on creation of the object which is then normally sent to the DBMS at that time.
- When our code wants to execute the statement – the DBMS does not have to compile it.
- We can re-use our `PreparedStatement` object without requiring re-compilation by the DBMS

PreparedStatement – Select Query

```
List<Customer> customers = new ArrayList<Customer>();

try {
        con = this.getConnection();

        String query = "SELECT * FROM CUSTOMER";
        ps = con.prepareStatement(query);

        rs = ps.executeQuery();
        while (rs.next()) {

                int customeNumber           = rs.getInt("CUSTNUMBER");
                String customerName         = rs.getString("CUSTNAME");
                String contactLastName      = rs.getString("CONTACTLASTNAME");
                String contactFirstName     = rs.getString("CONTACTFIRSTNAME");
                String phone                = rs.getString("PHONE");
                double creditLimit          = rs.getDouble("CREDITLIMIT");

                Customer cust = new Customer(customerNumber, customerName,
                        contactLastName, contactFirstName, phone, creditLimit);

                customers.add(cust);
   }
} catch (SQLException e) {
        throw new DaoException("findAllCustomers()" + e.getMessage());
}
```

PreparedStatement – Update

```
con = getConnection();

String command = "UPDATE CUSTOMER SET CUSTNAME=?,
                  CONTACTLASTNAME=?, CONTACTFIRSTNAME=?,
                  PHONE=?, CREDITLIMIT=?
                  WHERE CUSTNUMBER=?";

ps = con.prepareStatement(command);
ps.setString(1, cust.getCustomerName());
ps.setString(2, cust.getContactLastName());
ps.setString(3, cust.getContactFirstName());
ps.setDouble(4, cust.getPhone());
ps.setDouble(5, cust.getCreditLimit());
ps.setInt(6, cust.getCustomerNumber);

rowsAffected = ps.executeUpdate();
```