Tiers (or layers)

Separation of concerns

Hiding the type of storage from the client class

Let’s say we have a program that needs to fetch objects from a storage.
Should the program have to be concerned with the specifics of the storage?

Hiding the type of storage from the client class

No!?

What is a storage to the client?

It’s some means of storing something - and means of retrieving or otherwise interacting with the storage thingy.

Abstraction

OK, so something is stored somehow and we need to access that something.
Let’s use abstraction - so that we can talk about the storage in general terms.

A storage abstraction

Let’s say we have stored data on students somewhere, and our program needs access to the stored students.

We’d like to be able to fetch all students, for instance.

Therefore we’d like to have a programming interface to some kind of storage which holds the data on our students.

In Java, this sounds a lot like an interface!

The StudentStorage interface

The program/client class needs to get a list of all students. Sometimes it needs to get one student, providing some kind of key.

Suggestion:

```
public List<Student> getAllStudents();

public Student getStudentById(int id);
```

Now, we (the client code) can get all students from storage, or get one student if we provide a student id...

Simple StudentStorage interface

```
package so.and.so;
import java.util.List;
import so.and.so.domain.Student;

public interface StudentStorage {
  public List<Student> getAllStudents();
  public Student getStudentById(int id);
}
```

All we needed was a way to get all students as a list, and a way to get one student by providing a student id. So that’s pretty much it! (Not including Exceptions)

But, that’s just an interface… Who’s implementing?

Someone can create a concrete class which implements the interface. Let’s pretend someone’s done that, using JDBC to get the actual data to create Student objects. They would use that to implement the only two methods specified by the interface; `getAllStudents()` and `getStudentById(int id)`.

How do we know what concrete classes exists? We can have a StudentStorageFactory that will give us some concrete class implementing the interface StudentStorage. We still wouldn’t have to care about what concrete class does that job for us, and how that job is done!

StudentStorageFactory

Something like this:

```
public class StudentStorageFactory {
  public static StudentStorage getStorage() {
    //find out what storage to use and return for instance:
    return new StudentStorageDB();
  }
}
// StudentStorageDB is a class which implements
// StudentStorage and uses a database to accomplish that
```

What’s the point?

Well, the point is that now we have a means to get a list of students without having to know or care about how those students are stored. We have a programming interface (using a Java interface) that we can use!

In our client code (like our program’s main method for instance) which can simply do this:

```
StudentStorage storage = StudentStorageFactory.getStorage();
```

Classes/interfaces needed so far...

Our Main class that runs the program

A StudentStorage interface giving us the methods we need

A StudentStorageFactory class giving us some implementation of the interface

A StudentStorageDB class, implementing the StudentStorage interface using a database.

Example code in our client class

Our main method could now look like this (not showing exceptions):

```
public static void main(String[] args) {
  StudentStorage storage = StudentStorageFactory.getStorage();
  List<Student> students = storage.getAllStudents();
  for(Student student : students) {
    // Do something with each Student...
  }
}
```

The point being? That we can access all students from the storage, without having to care about how those students are stored. Nothing in our code reveals how students are stored in the storage. Our code can focus on getting students and using all the Student objects...

So how would we implement the interface?

The implementation StudentStorageDB obviously fetches students from a database. So we'd need to use JDBC.

To keep things simple for the StudentStorageDB class, we could create a DBUtils helper class that handles the database setup.

DBUtils can set up the connection and so forth and provide a method for performing a query based on a String with SQL and returning a ResultSet with the result.

The DBUtil class

This class should load the JDBC driver and create a Connection to the database. It should expose/offer a method that given a String with SQL returns a ResultSet.

Signature could be:

```
public ResultSet query(String sql) {
  // code to create a statement and execute a query
}
```

What’s the gain for the StudentStorageDB class?

The StudentStorageDB class now only has to handle ResultSet objects (and doesn’t have to care about setting up the connection, loading the driver etc etc).

Code could be something like this (leaving out try-catch and other “details”):

```
public List<Student> getAllStudents() {
  ResultSet rs = db.query(SELECT_ALL_QUERY);
  List<Student> students = new ArrayList<Student>();
  while(rs.next()) {
    students.add( new Student(rs.getString("id"),
                              rs.getString("name")) );
  }
  return students;
}
```

What’s the gain for the StudentStorageDB class?

The code could for the getStudentById() could be something like this (leaving out try-catch and other details):

```
public Student getStudentById(int id) {
  String query = SELECT_STUDENT_QUERY + id;
  ResultSet rs = db.query(query);
  rs.next();
  Student student = new Student(rs.getString("id"),
                                rs.getString("name")) );
  return student;
}

*) what to do if rs.next() returns false?
```

What did StudentStorageDB do?

StudentStorageDB offered a method to get a List<Student> without exposing where it got it from. It implements the StudentStorage interface so that clients can use it as any StudentStorage - they all have in common that they offer one method for fetching all students from some storage. It also offered a method to get one Student from a provided student id.

The StudentStorage accomplished this using the DBUtil class which provided access to the database where all student data was stored.

The DBUtil could be implemented as a Singleton.

Overview

Main - create a StudentStorage from StudentStorageFactory

StudentStorageFactory selects a suitable implementation and returns an instance.

Main uses the storage to fetch all students and does something with them.

The implementation StudentStorageDB uses DBUtils to get a ResultSet from a query string, and creates a list of all students and returns the list.

DBUtils does the dirty work of setting up the DB-connection and statements needed to e.g. fetch students in a ResultSet which is returned.

Main

```
package org.henrikard.student.main;
import org.henrikard.student.storage.StudentStorageFactory;
import org.henrikard.student.storage.StudentStorage;
import org.henrikard.student.storage.StorageException;
import org.henrikard.student.domain.Student;
import java.util.List;

public class Main {
    public static void main(String[] args) {
      try{
          StudentStorage storage = StudentStorageFactory.getStorage();
          List<Student> students = storage.getAllStudents();
          for(Student student : students) {
            System.out.println(student);
          }
      } catch (StorageException se) {
          System.err.println("Error accessing the storage: "
                          + se.getMessage());
      }
    }
}
```

StudentStorage

```
package org.henrikard.student.storage;
import org.henrikard.student.domain.Student;
import java.util.List;

public interface StudentStorage {
    public List<Student> getAllStudents() throws StorageException;
    public Student getStudentById(int id) throws StorageException;
}
```

StudentStorageFactory

```
package org.henrikard.student.storage;

public class StudentStorageFactory {
    public static StudentStorage getStorage() {
      // We only have one type of storage just now...
      return new StudentStorageDB();
    }
}
```

Student

```
package org.henrikard.student.domain;

public class Student {
    private String name;
    private int id;
    public Student(int id, String name) {
      this.name = name;
      this.id   = id;
    }
    public String name(){   return this.name; }
    public int id(){  return id; }
    @Override
    public String toString() {
      return new StringBuilder().append(name)
          .append(" (")
          .append(id)
          .append(")")
          .toString();
    }
}
```

StorageException

```
package org.henrikard.student.storage;

public class StorageException extends Exception {
    public StorageException(String msg) {
      super(msg);
    }
}
```

DBUtil pt1

```
package org.henrikard.student.db;

import java.sql.*;

public class DBUtil {

    private static Connection con;
    private static final String DB_CON_STR =
              "jdbc:sqlite:org/henrikard/student/resources/students.db";
    static {
        try {
            Class.forName("org.sqlite.JDBC");
            con = getConnection();
        } catch (ClassNotFoundException cnfe) {
            System.err.println("Could not load database driver: "  +
                                   cnfe.getMessage());
        }
    }
```

DBUtil pt2

```
private static Connection getConnection() {
     if (con == null) {
         try {
             con = DriverManager.getConnection(DB_CON_STR);
         } catch(SQLException sqle) {
             System.err.println("Error creating a connection to the database: " +
                                   sqle.getMessage());
         }
     }
     return con;
 }

 private Statement statement() throws SQLException {
     return con.createStatement();
 }
```

DBUtil pt3

```
    private static DBUtil instance;

    private DBUtil() {
    }

   public static DBUtil getInstance() {
        if (instance == null) {
            instance = new DBUtil();
        }
        return instance;
    }

    public ResultSet query(String sql) throws SQLException {
        Statement stm = statement();
        ResultSet rs = stm.executeQuery(sql);
        return rs;
    }
}
```

What did we gain from all this?

We separated concerns so that:

Main only fetches students and operates on them

StudentStorage only specifies what a storage is and can do

StudentStorageDB implements StudentStorage using a ResultSet

DBUtils handles all database stuff and only exposes ResultSet

Compile and run

```
# change directory to the one above org

javac */*/*/*/*.java && java -cp .:org/henrikard/student/resources/sqlite-jdbc-3.8.11.2.jar
                    org.henrikard.student.main.Main

(on one single row)
```

Directory and file layout

```
.
`-- org
    `-- henrikard
        `-- student
            |-- db
            |   `-- DBUtil.java
            |-- domain
            |   `-- Student.java
            |-- main
            |   `-- Main.java
            |-- resources
            |   |-- sqlite-jdbc-3.8.11.2.jar
            |   `-- students.db
            `-- storage
                |-- StorageException.java
                |-- StudentStorageDB.java
                |-- StudentStorageFactory.java
                `-- StudentStorage.java

8 directories, 9 files
```