## Introduction

This post builds on the programming values set out in the last post.

## The database interface

In the last post, I defined the following interface:

interface IDatabase {

function query($sql);

function execute($sql);

function executeBatch($list);

function getInsertID();

function close();

}

I’ve now added some extra methods. Three of them relate to database transactions, which I explain below. The other two relate to prepared statements, which we’ll cover later in the course when we look at SQL injection attacks. Here are the new methods:

|  |  |
| --- | --- |
| *Method* | *Functionality* |
| beginTransaction() | Tells the database we want to start a transaction. |
| commitTransaction() | Tells the database we’ve finished the transaction. |
| rollbackTransaction() | Tells the database we want to undo all the changes we’ve made so far in the current transaction. |
| queryPrepared(  $paramaterisedSQL,  $fields) | This method will substitute fields into a parameterised SQL statement in a way that protects against SQL injection and run the query. |
| executePrepared(  $paramaterisedSQL,  $fields) | This method will substitute fields into a parameterised SQL statement in a way that protects against SQL injection and execute the statement |

As before, if anything goes wrong, an exception will be thrown.

## Transactions

Databases have so called ***acid*** properties. (see <http://en.wikipedia.org/wiki/ACID>). One of the key goals is that the database goes from one valid state to another in a single indivisible step; this it ***atomicity*** (the letter ‘**a’** in acid). Because there may be many concurrent users of a database, it is also important that intermediate steps are ***isolated*** from other users (the letter **i** in acid).

For example, let’s say we have an orders table and an order details table; the orders table has general information about the order, such as date, customer, delivery address and the details table has several rows for each order each of which has an order id, product code and quantity. To add an order, we’ll need to create the orders table row first, ask the database for its ID if it is auto allocated and then add each of the order lines, using that ID. This will take several SQL statements. However, we don’t want another user to see a part completed order – the order should appear magically in its fully complete state, or not at all.

We do this by using a database transaction. We use ***beginTransaction*** to start it off. We can then interact with the database and the database will isolate the relevant parts of the database, typically by using locks to keep others out while the transaction is being processed. We end the transaction by telling the database to ***commit*** the changes. This releases the changes to the world in a single indivisible operation. If we find a problem as we’re working through the transaction SQL statements, we can roll the transaction back so that it is as if the transaction never happened.

## SOLID values

The letter O in solid refers to the open-closed principle. This states that an object should be open to extension but closed to modification. I’ve deliberately released the database logic in two stages to demonstrate this principle. First, under the open-closed principle, what I should have done would be to define another interface IDatabaseExtended that inherited from IDatabase and extended it to include the extra methods. Had I done this, I would have broken no code, but, of course, all the code would have used the old definitions.

Instead, I chose to modify IDatabase by adding the extra methods. This immediately breaks the old implementation so I had to update the Database class to implement the new Interface. However, I did not feel this was too big a task and refactoring is part of life for an agile developer. Perhaps more important is that because I was only adding to the interface, the change won’t affect any code that uses the interface. That will still work and won’t need refactoring.

If you look at the code, you’ll see I have fully implemented the new methods dealing with transactions, but for those dealing with prepared statements, my implementation just throws an exception “Not yet implemented”. This is quite a common pattern with “just-in-time” architectural design. I know I’ll need it, but I don’t need to implement it yet. If I forget, the exception will soon remind me when I come to use it! More importantly, I’ll implement it when I need it – when I’m working on the use case that requires it. This means the use case will be fresh in my mind when I write the implementation.

Another point about the open/closed principle is that when we say closed to modification, we mean close to modification of the public API – the interface. I’ve taken the opportunity to tidy up some of the implementation of the previous version, for example by moving the error message creation to a shared function. I’m always free to change the internal implementation of any class, provided I don’t change the public API or semantics; these changes don’t count as a modification for the SOLID values.

## Exceptions and PHP version

Exceptions were only brought into PHP with version 5. Previous versions of PHP only supported error codes. Exceptions let us write much cleaner and more robust code – make them your friend. Similarly, OO technology was only introduced in version 5. That means we need to make sure that we’re working on at least version 5. In this framework design, I’m assuming version 5.3 or later. Version 5.3, although not the latest, is still widely used and choosing a later version would severely limit the range of hosting companies I could use. One common mistake beginning web site developers make is to always choose the latest version and then find they can’t host their web site!

I’ll be adding a version check in the production code to make sure it’s running on an appropriate version, but for the moment, I’ll just leave that as something to do later. (It’s quite easy!)

## Testing

## I’ve updated the test database script to exercise the new database methods.