## Code review

This document carries out a self-evaluated code review of my categories model (see appendix A). The review identifies the extent to which the code demonstrates our core programming values and good OO design. The core values are

* Don’t repeat yourself
* Separation of concerns

The criteria for good OO design are.

* **S**ingle responsibility,
* **O**pen-closed,
* **L**iskov substitution,
* **I**nterface segregation and
* **D**ependency inversion

The value ***don’t repeat yourself*** is demonstrated by the use of a hierarchy AbstractModel – AbstractEntityModel -> concrete CategoryModel. The AbstractModel contains all logic that applies to models in general. The AbstractEntityModel contains logic that applies to models that are associated with a database table. Thus, methods such as getting or checking an ID belong here. This model implements the core logic required for any entity and uses abstract functions to define the things a concrete subclass must override. The category model contains only those things specific to a category entity. This is:

|  |  |
| --- | --- |
| Section one | Private instance data for name and description. |
| Section two | Getters and setters for private instance data  Setters validate data before setting and notify changes via the parent’s **didChange**() method |
| Section three | Implementation of must overrides (abstract functions). These are:  **init()** – set all required fields to null  **allRequiredFieldsArePresent()** – checks all required fields have been specified  **load($id**) – loads private data from the database  **getInsertionSql()** – supplies the SQL needed to add a new category  **getUpdateSql()** – supplies the SQL needed to update the category  **getDeletionSql()** – supplies the SQL needed to delete the category |
| Section four | Shared (static) validation routines (**errorInName** and **errorInDescription**) which use a helper (**errorInRequiredField**) in the superclass.  A shared (static) function **isExistingId** that checks if there is already a category with a specified ID. This uses a helper function **checkExistingId** in the superclass. |

The class demonstrates **single responsibility** as follows. It provides full CRUD capability and checks all aspects of category data for validity before it actions any changes. Exceptions are thrown if anything is amiss. Thus, the class can *guarantee* the validity of the category data it manages and is a “one stop shop” for anything to do with categories. The **open-closed principle** (open to extension, closed to modification) is demonstrated, first, by the refactoring process which produced the abstract model (see the refactoring screencast I posted), and second by the fact that extended capability could be created by sub-classing the CategoryModel, overriding the methods in section three, and adding new methods for the extended capability. **Liskov substitution** is demonstrated, both by the inheritance hierarchy used in the refactoring process and by the fact that it would be possible to use both the current category model and its new subclass in the extension just described. **Interface segregation** is demonstrated by the overall architecture. Although the front-end-controller supplies a comprehensive context, this is segregated into separate interfaces for URI, session, database, etc. Since a model should know nothing about its context, this means that a model just needs the IDatabase interface, not the others. Finally, the model is not dependent on any database technology – it will work with any object that supports the IDatabase interface. The object is supplied in the constructor. This demonstrates **dependency inversion** – the dependency is *injected* in the constructor.

**Separation of concerns** is demonstrated throughout. The class implements everything required for CRUD of category data and nothing else. All other concerns are delegated elsewhere.

## Appendix A

|  |
| --- |
| <?php  /\*  A PHP framework for web sites by Mike Lopez    A sample entity model for a product category  ============================================  Design features:  \*\* Static methods (ErrorIn...) can be used to check fields before creation  \*\* anything invalid after creation throws an exception  \*\* Creating with an ID loads from the database  \*\* after setting data, a save will update the database  \*\* delete will remove the category from the database and clear all data    NB if cascade delete is not used, deletion will fail if there are dependent  entities in the database  \*/  include 'lib/abstractEntityModel.php';  ;  class CategoryModel extends AbstractEntityModel {  /\* Section one  ===========    Private instance data  \*/  private $name;  private $description;  // standard constructor  public function \_\_construct($db, $id=null) {  parent::\_\_construct($db,$id);  }  /\* Section two  ===========    Getters and setters for private instance data  Setters validate data before setting and notify changes via  didChange()  \*/  public function getName() {  return $this->name;  }  public function getDescription() {  return $this->description;  }  public function setName($value) {  $this->assertNoError($this->errorInName($value));  $this->name=$value;  $this->didChange();  }  public function setDescription($value) {  $this->assertNoError($this->errorInDescription($value));  $this->description=$value;  $this->didChange();  }  /\* Section three  ===========    Implementation of all must overrides  These modify the generic logic in the abstract model to define  the specific logic that applies to categories  \*/  // set default values for instance data  // (required fields should be set to null)  protected function init() {  $this->name=null;  $this->description=null;  }  // load instance data from database  protected function load($id) {  $sql="select name, description from categories ".  "where categoryID = $id";  $rows=$this->getDB()->query($sql);  if (count($rows)!==1) {  throw new InvalidDataException("Category (ID $id) not found");  }  $row=$rows[0];  $this->name=$row['name'];  $this->description=$row['description'];  }  // return false if any required field is null  protected function allRequiredFieldsArePresent() {  return $this->name !== null &&  $this->description !==null;  }  // sql to insert instance data into database  protected function getInsertionSql() {  $name=$this->safeSqlString($this->name);  $description=$this->safeSqlString($this->description);  return "insert into categories(name, description) values ($name, $description)";  }  // sql to update database record from instance data  protected function getUpdateSql() {  $name=$this->safeSqlString($this->name);  $description=$this->safeSqlString($this->description);  return "update categories set ".  "name=$name, ".  "description=$description ".  "where categoryID=".$this->getId();  }  // sql to delete this instance  protected function getDeletionSql() {  return 'delete from categories where categoryID = '.$this->getId();  }    /\* Section four  ===========    Validation functions for all fields  These are all static (shared class level) functions  \*/  public static function errorInName($value) {  return self::errorInRequiredField('Category name',$value,40);  }  public static function errorInDescription($value) {  return self::errorInRequiredField('Description',$value,200);  }  public static function isExistingId($db,$id) {  return self::checkExistingId($db,$id,  'select 1 from categories where categoryID='.$id);  }  } |