Web Application Design

Chapter 14

Objectives

1. Real World Web Software Design
2. Principle of Layering
3. Design Patterns in Web Context
4. Data and Domain Patterns
5. Presentation Patterns

Section 1 of 5

REAL WORLD WEB SOFTWARE DESIGN

Real-World Software Design

Software design can mean many things.

In general, it is used to refer to the planning activity that happens between gathering requirements and actually writing code.

This chapter has an overview of some of the typical approaches used in the software design of web applications

Challenges

In designing real-world applications

it is quite possible to create complex web applications with little to no class design.

The **page-oriented development approach** sees each page contain most of the programming code it needs to perform its operations.

For sites with few pages and few requirements, such an approach is quite acceptable.

Challenges

In designing real-world applications

Real software projects are notoriously vulnerable to shifting requirements; web projects are probably even more so.

- New features will be added and other features will be dropped.
- The data model and its storage requirements will change.
- The execution environment will change from the developers' laptops to a testing server, a production server, or perhaps a farm of web servers

Section 2 of 5

PRINCIPLE OF LAYERING

Challenges

In designing real-world applications

It is in this type of web development environment that rapid ad-hoc design practices may cause more harm than benefit,

Rapidly thought-out systems are rarely able to handle unforeseen changes in an elegant way.

For these reasons, many web developers make use of a variety of software design principles and patterns

Layering

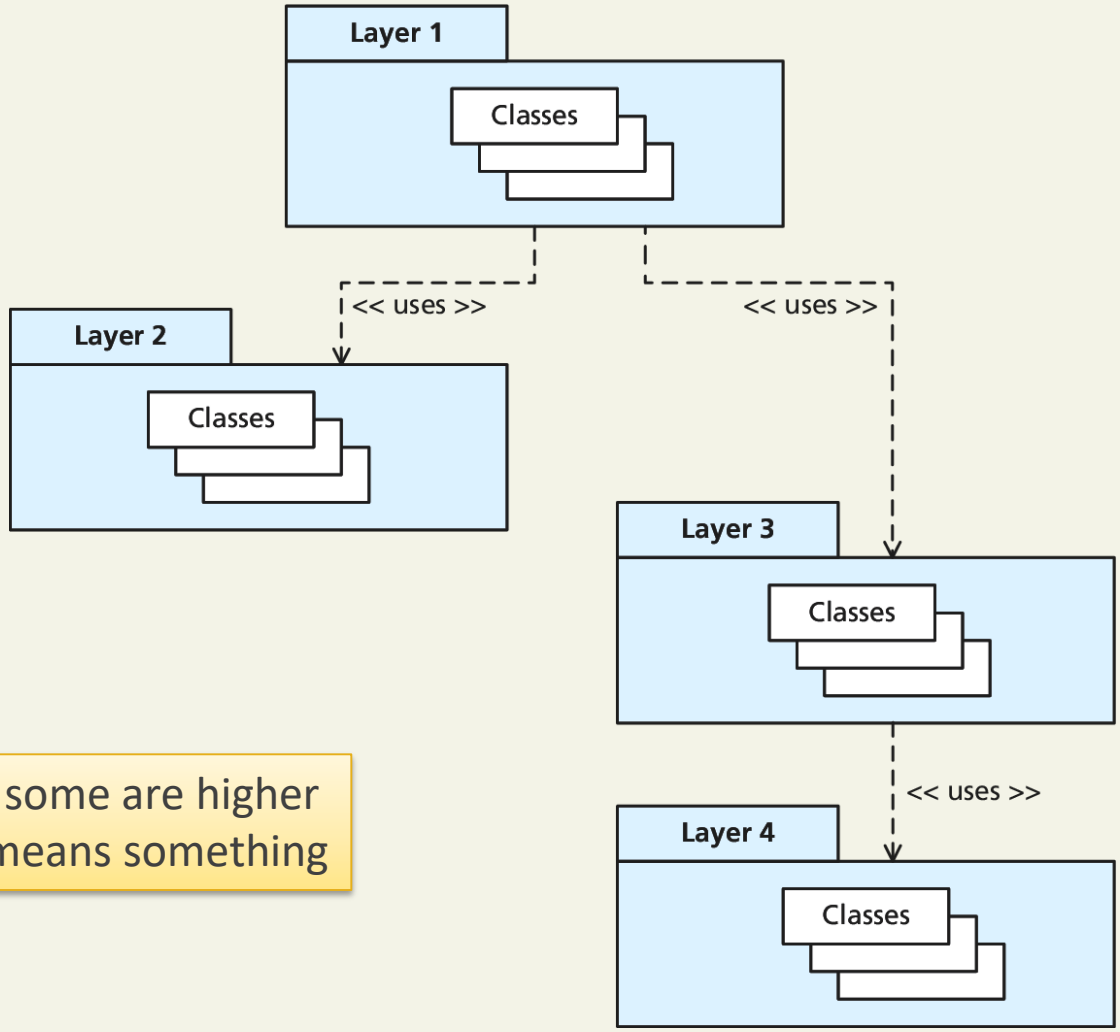
Break apart a complicated system

A **layer** is simply a group of classes that are functionally or logically related; that is, it is a conceptual grouping of classes.

- Each layer in an application should demonstrate **cohesion** (how much they belong together)
- The goal of layering is to distribute the functionality of your software among classes so that the **coupling** of a given class to other classes is minimized.
- A **dependency** is a relationship between two elements where a change in one affects the other.

Layering

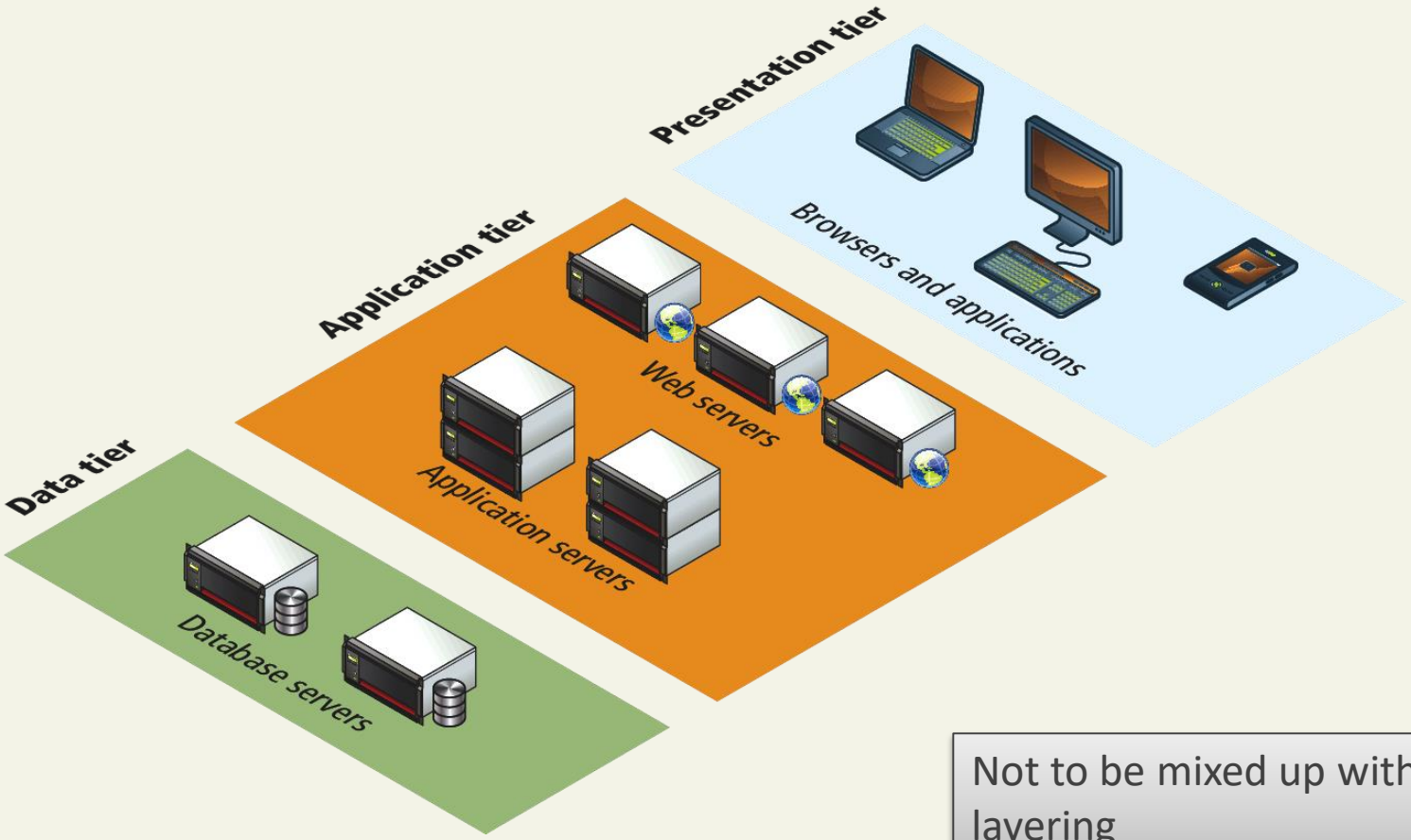
See the relationships



The fact that some are higher than others means something

Tiers

A **tier** refers to a processing boundary



Not to be mixed up with layering

Layers

Benefits

- The application should be more maintainable and adaptable to change since the overall coupling in the application has been lowered
- When an application has a reliable and clearly specified application architecture, much of the page's processing will move from the page to the classes within the layers.
- A given layer may be reusable in other applications, especially if it is designed with reuse in mind

Layers

Disadvantages

- The numerous layers of abstraction can make the resulting code hard to understand at first
- the extra levels of abstraction might incur a small performance penalty at run time

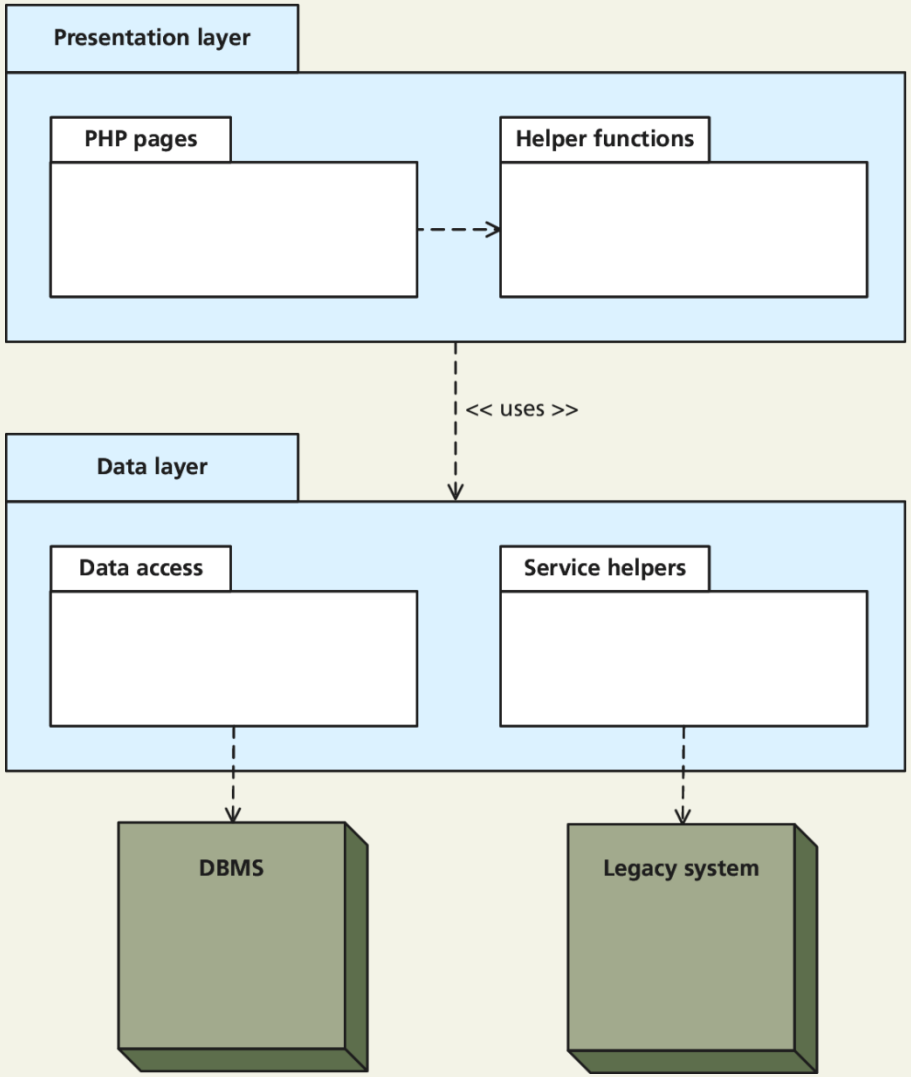
Common Layering Schemes

Principle Software Layers

- **Presentation** Principally concerned with the display of information to the user, as well as interacting with the user.
- **Domain/Business** The main logic of the application. Some developers call this the business layer since it is modeling the rules and processes of the business for which the application is being written.
- **Data Access** Communicates with the data sources used by the application. Often a database, but could be web services, text files, or email systems. Sometimes called the technical services layer.

Two Layer Model

A Common Layering Scheme



Two Layer Model

A Common Layering Scheme

The advantage of the two-layer model is that it is relatively easy to understand and implement.

In a two-layer model, each table typically will have a matching class responsible for **CRUD** (create, retrieve, update, and delete) functionality for that table.

The drawbacks of the two-layer model is its hard to implement business rules and processes.

** I envision that we'll be using this for our projects.

If there is some "business rules" we can implement case-by-case

Business Rules

What are they?

A **business rule** refers not only to the usual user-input validation and the more complex rules for data that are specific to an organization's methods for conducting its business.

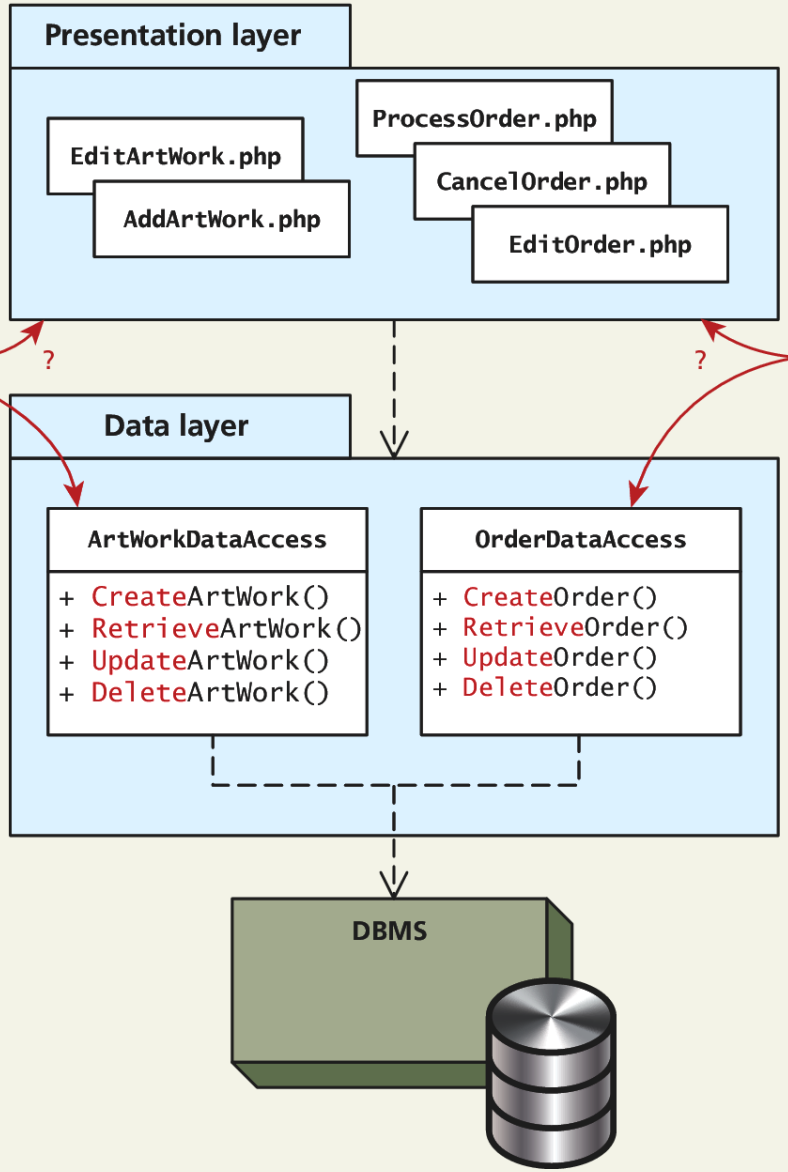
Do they belong within the PHP of the order form?

Do they belong instead in the data access layer? (implemented in the database or sql call)

Business Rules

Where do they go?

Case where they sell works of art



Business Rules

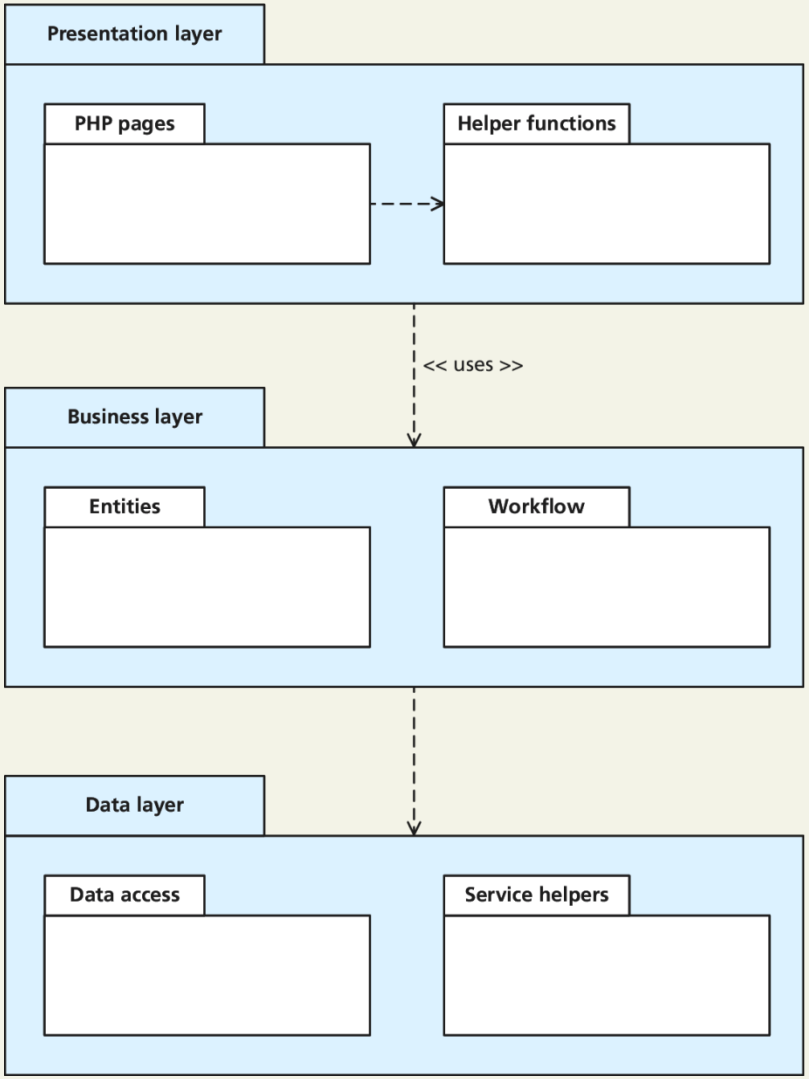
- When creating an art work title, ensure that it doesn't already exist.
- Only allow delete if no orders yet for this art work.
- Ensure price is greater than cost.

Business Processes

- After creating order, check if it qualifies for any discounts.
- Check if selected shipper available for weight of order.
- Ensure financial system approved purchase.
- Communicate to inventory system to fulfill (ship) order.
- Only allow order to be canceled if inventory system has not fulfilled order.
- Communicate with financial system to refund purchase (or get more funds if necessary).

Business Rules

Add another layer



Business Rules

In the middle layer

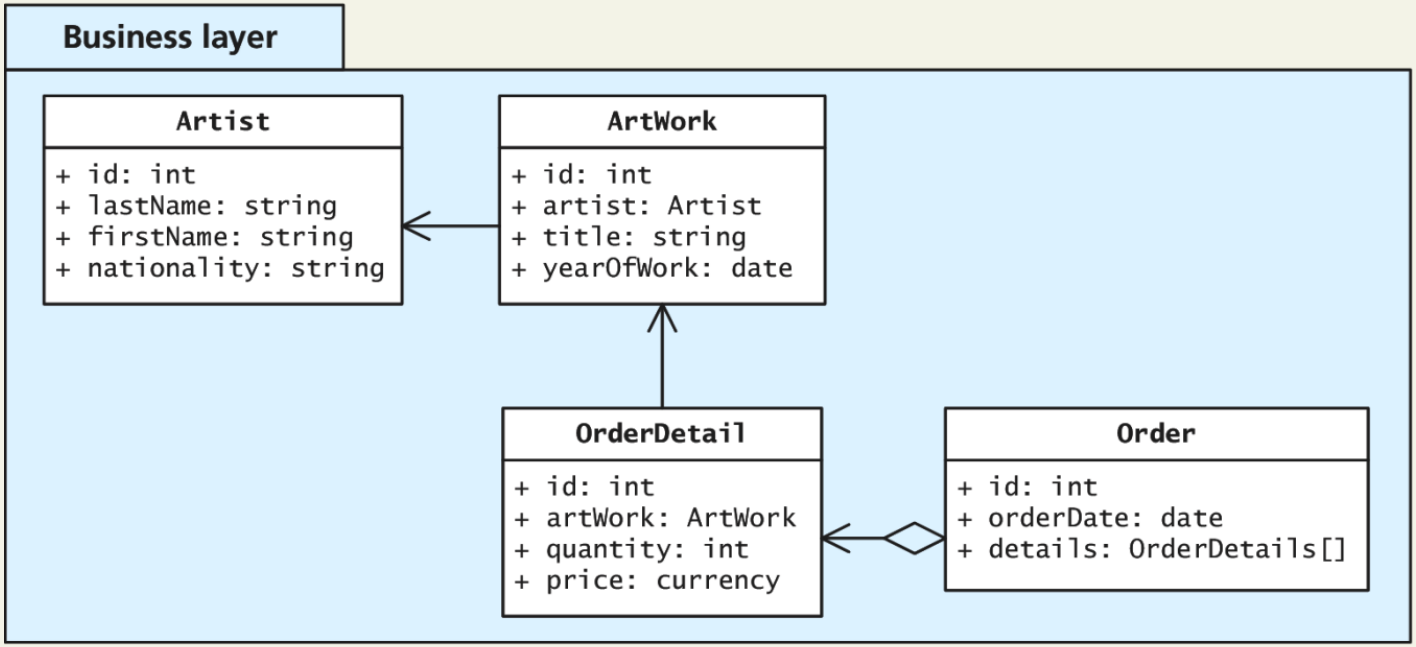
Classes within the “middle” layer of a three-layer model as **business objects**;

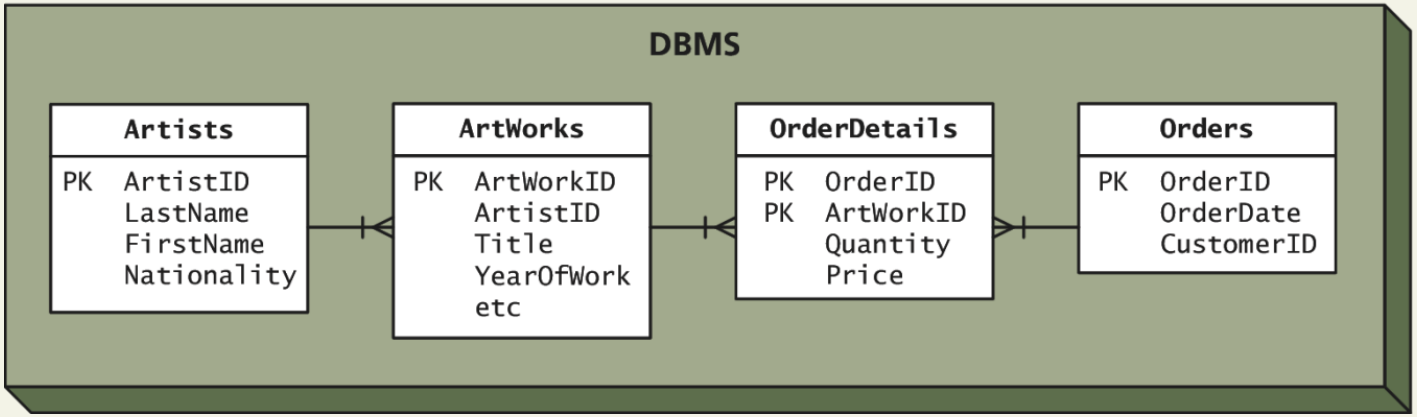
Also **entities** or **domain objects**.

Regardless of what they are called, business objects represent *both* the data and behavior of objects that correspond to the conceptual domain of the business.

Business Layer

The middle layer

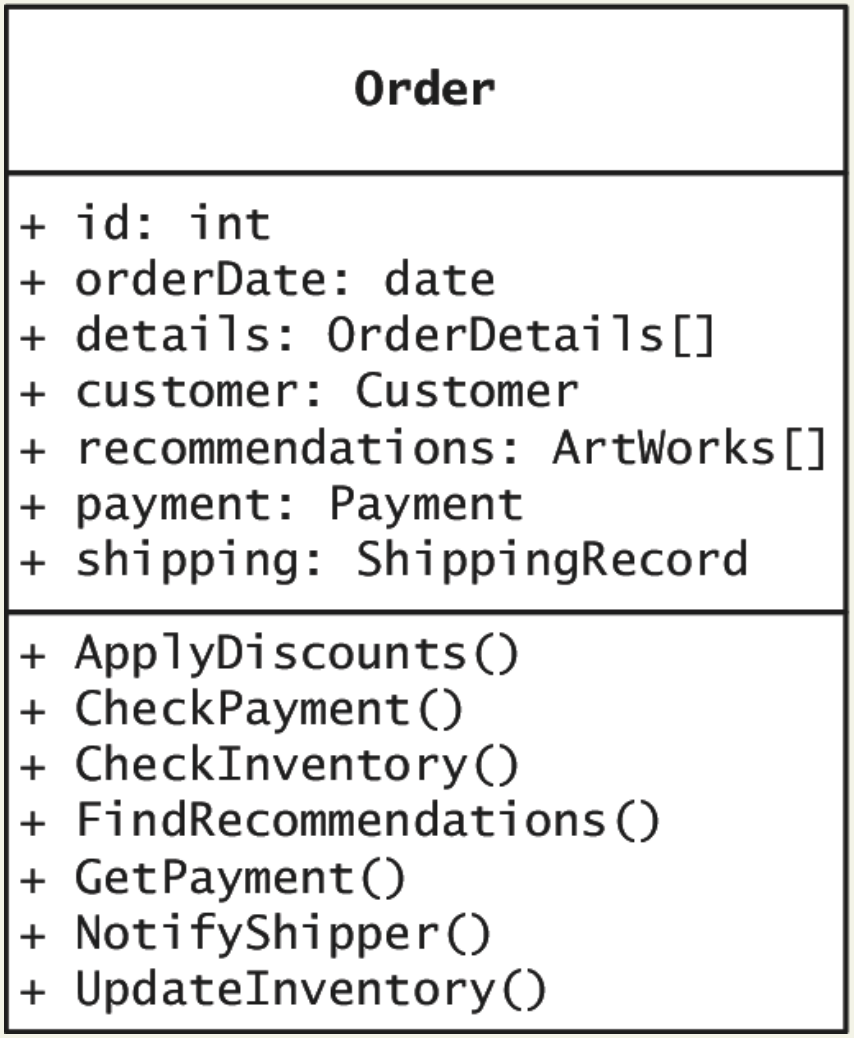




Business Layer

An example complex layer

Remind you of actually shopping?



Section 3 of 5

DESIGN PATTERNS IN THE WEB CONTEXT

Software Design Patterns

Same old problem

What are design patterns?

Over time as programmers repeatedly solved whole classes of problems, consensus on best practices emerged for how to design software systems to solve particular problems.

These best practices were generalized into reusable solutions that could be adapted to many different software projects. They are commonly called **design patterns**, and they are useful tools in the developer's toolbox.

Software Design Patterns

Will not solve all your problems, but they will help you design better code if used thoughtfully.

It is a clear and concise way to describe algorithms/code

The most common design patterns are those that were identified and named in the classic 1995 book *Design Patterns: Elements of Reusable Object-Oriented Software*

Adapter Pattern

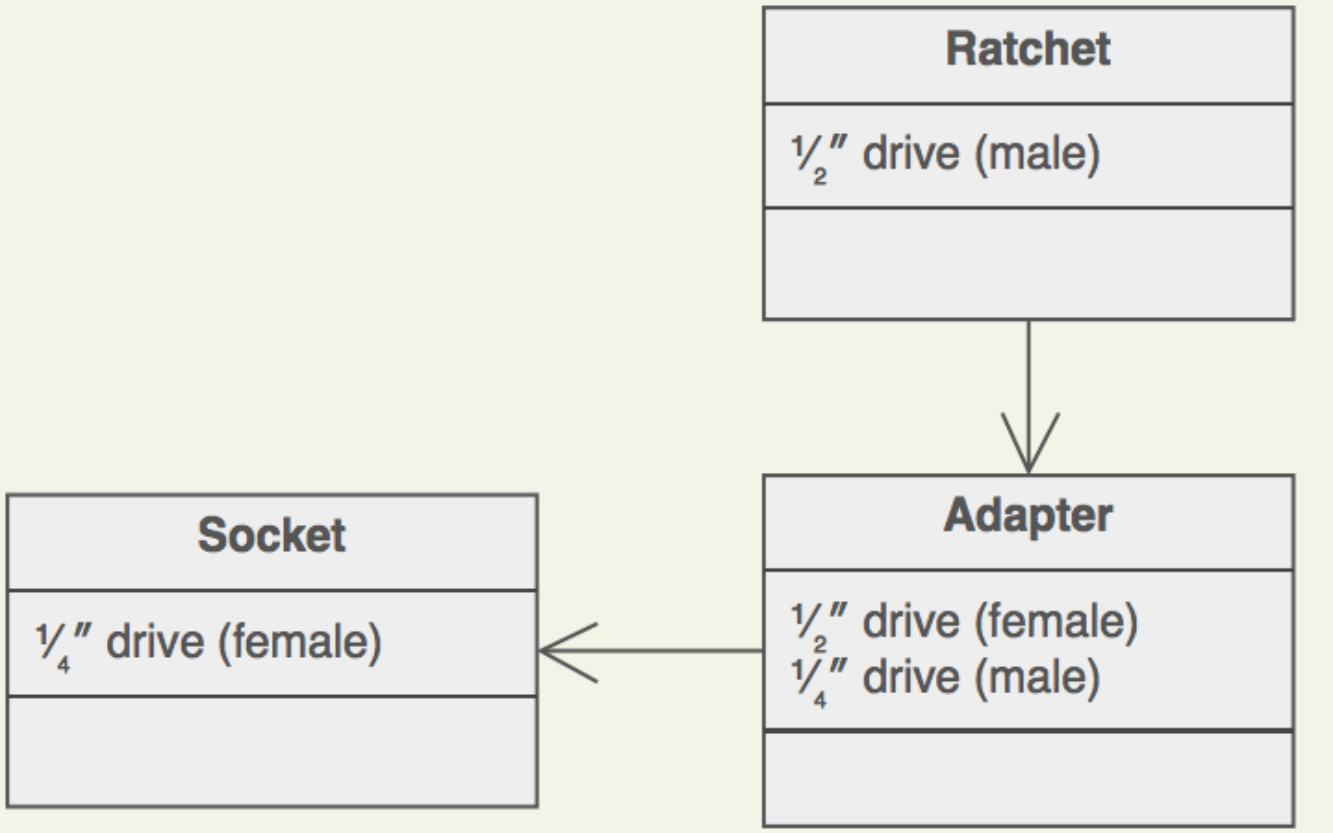
The **Adapter pattern** is used to convert the interface of a set of classes that we need to use to a different but preferred interface.

The Adapter pattern is frequently used in web projects as a way to make use of a database API (such as PDO or mysqli) without coupling the pages over and over to that database API.

PDO – PHP Data Objects

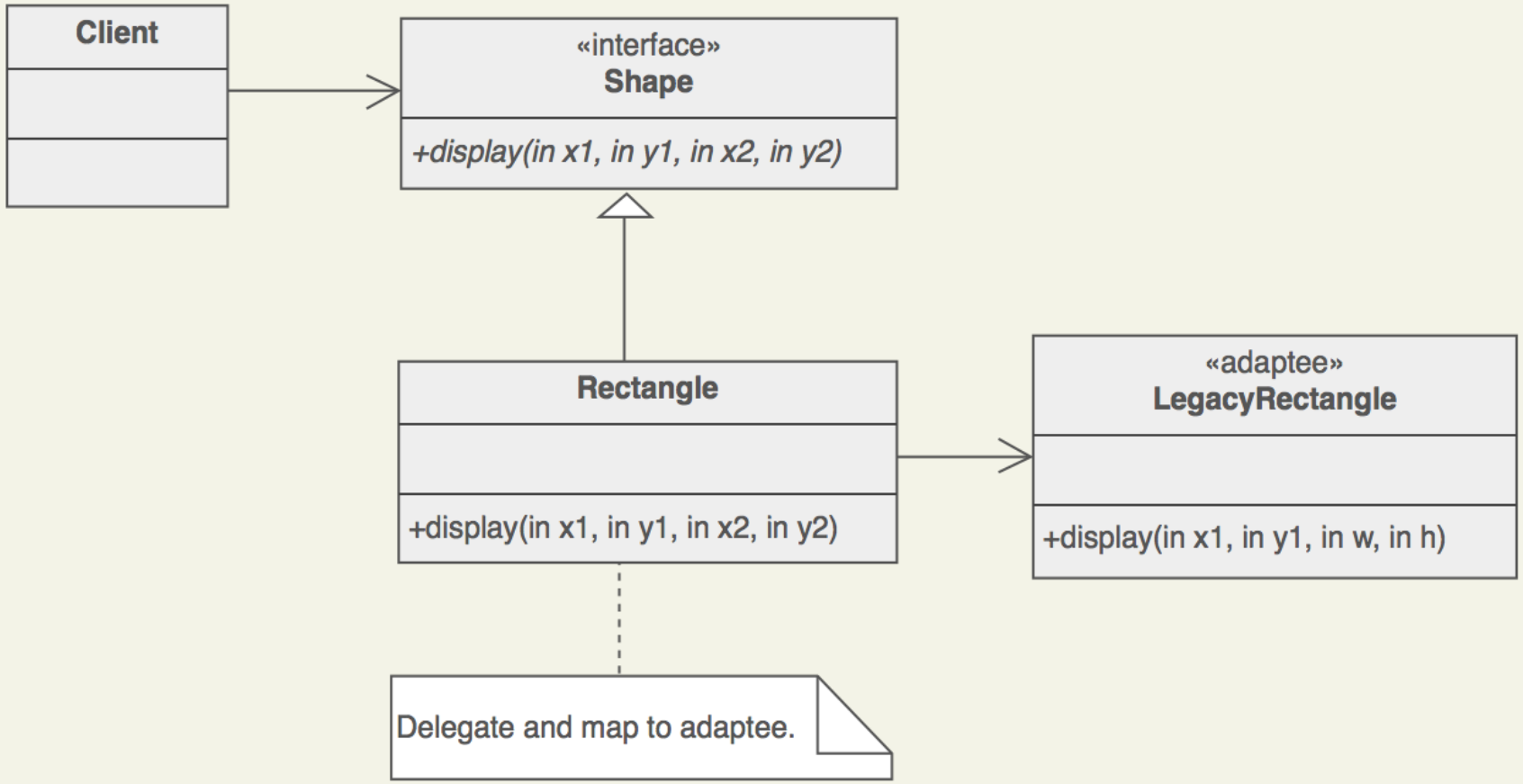
mysqli – mysql improved

Adapter Pattern



Adapter Pattern

Below, a legacy Rectangle component's display() method expects to receive "x, y, w, h" parameters. But the client wants to pass "upper left x and y" and "lower right x and y". This incongruity can be reconciled by adding an additional level of indirection – i.e. an Adapter object.

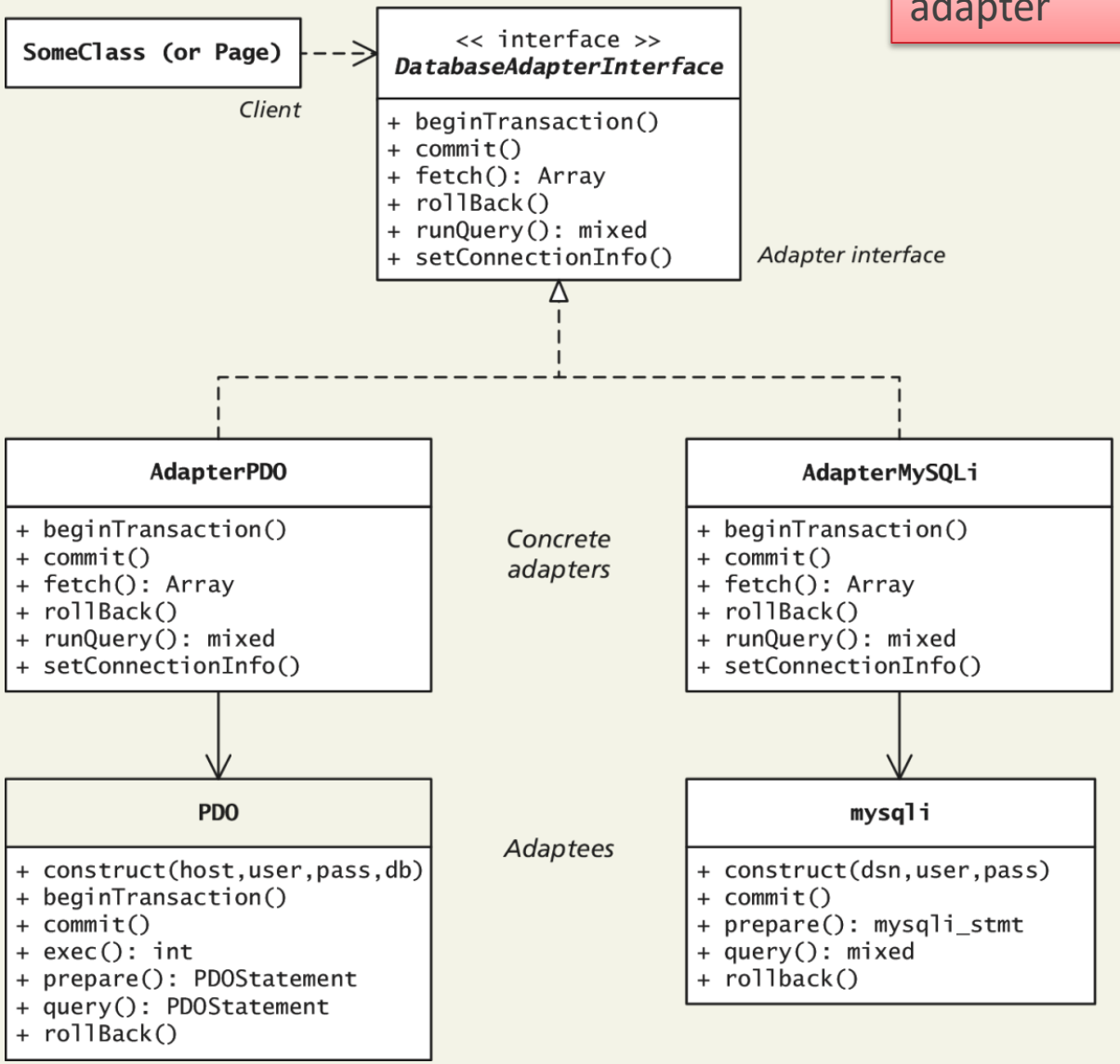


http://sourcemaking.com/design_patterns/adapter

Adapter Pattern

UML Diagram

For the database adapter



Adapter Pattern

Interface for adaptor

```php
<?php
/*
  Specifies the functionality of any database adapter
*/
interface DatabaseAdapterInterface
{
    function setConnectionInfo($values=array());
    function closeConnection();

    function runQuery($sql, $parameters=array());
    function fetchField($sql, $parameters=array());
    function fetchRow($sql, $parameters=array());
    function fetchAsArray($sql, $parameters=array());

    function insert($tableName, $parameters=array());
    function getLastInsertId();
    function update($tableName, $updateParameters=array(),
                    $whereCondition='', $whereParameters=array());
    function delete($tableName, $whereCondition=null,
                    $whereParameters=array());
    function getNumRowsAffected();

    function beginTransaction();
    function commit();
    function rollBack();
}

?>
```

LISTING 14.1 Interface for adapter

Adapter Pattern

Concrete Classes (partial implementation)

```php
<?php
/*
  Acts as an adapter for our database API so that all database API
  specific code will reside here in this class. In this example, we
  will use the PDO API.
*/
class DatabaseAdapterPDO implements DatabaseAdapterInterface
{
   private $pdo;
   private $lastStatement = null;

   public function __construct($values) {
      $this->setConnectionInfo($values);
   }

   /*
    Creates a connection using the passed connection information
   */
   function setConnectionInfo($values=array()) {
      $connString = $values[0];
      $user = $values[1];
      $password = $values[2];

      $pdo = new PDO($connString,$user,$password);
      $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
      $this->pdo = $pdo;
   }

   /*
    Executes a SQL query and returns the PDO statement object
   */
   public function runQuery($sql, $parameters=array()) {
```

Adapter Pattern

Concrete Classes (partial implementation)

Any client classes (or pages) that needs to make use of the database will do so via the *concrete adapter*:

```
$connect = array(DBCONNECTION, DBUSER, DBPASS);

$adapter = new DatabaseAdapterPDO($connect);

$sql = 'SELECT * FROM ArtWorks WHERE ArtWorkId=?';

$results = $adapter->runQuery($sql, array(5));
```

This code sample contains a dependency via the explicit instantiation of the DatabaseAdapterPDO class. If you at some point switch to a different adapter, you will need to change every instantiation to the appropriate concrete adapter.

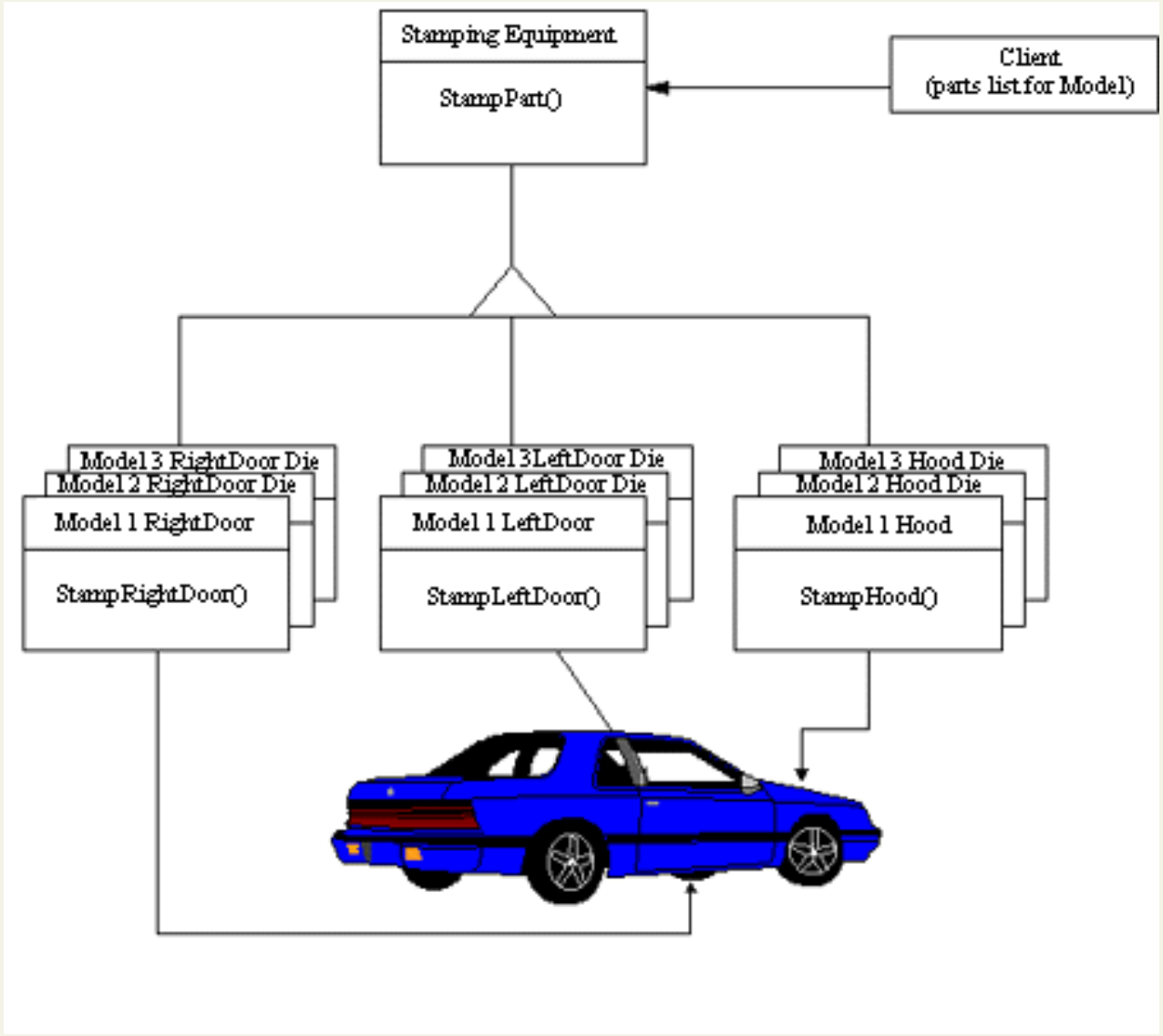
Simple Factory

Addresses the dependency of Adaptor

A **factory** is a special class that is responsible for the creation of subclasses, so that clients are not coupled to specific subclasses or implementations.

Since PHP is a late-binding language (type at run time), you can create a factory class that avoids conditional logic by dynamically specifying at run time the specific class name to instantiate

```
$adapter = DatabaseAdapterFactory::create('PDO', $connectionValues);

$results = $adapter->runQuery('SELECT * FROM Artists');
```


Stamping Equipment
StampPart()
Client
(parts list for Model)
Model 3 RightDoor Die
Model 2 RightDoor Die
Model 1 RightDoor
StampRightDoor()
Model 3 LeftDoor Die
Model 2 LeftDoor Die
Model 1 LeftDoor
StampLeftDoor()
Model 3 Hood Die
Model 2 Hood Die
Model 1 Hood
StampHood()

Simple Factory

Addresses the dependency of Adaptor

```php
<?php
/*
  An example of a Factory Method design pattern. This one is
  responsible for instantiating the appropriate data adapter
*/
class DatabaseAdapterFactory {
    /*
     Notice that this creation method is static. The $type parameter
     is used to specify which adapter to instantiate.
    */
    public static function create($type, $connectionValues) {
        $adapter = "DatabaseAdapter" . $type;
        if ( class_exists($adapter) ) {
            return new $adapter($connectionValues);
        }
        else {
            throw new Exception("Data Adapter type does not exist");
        }
    }
}
?>
```

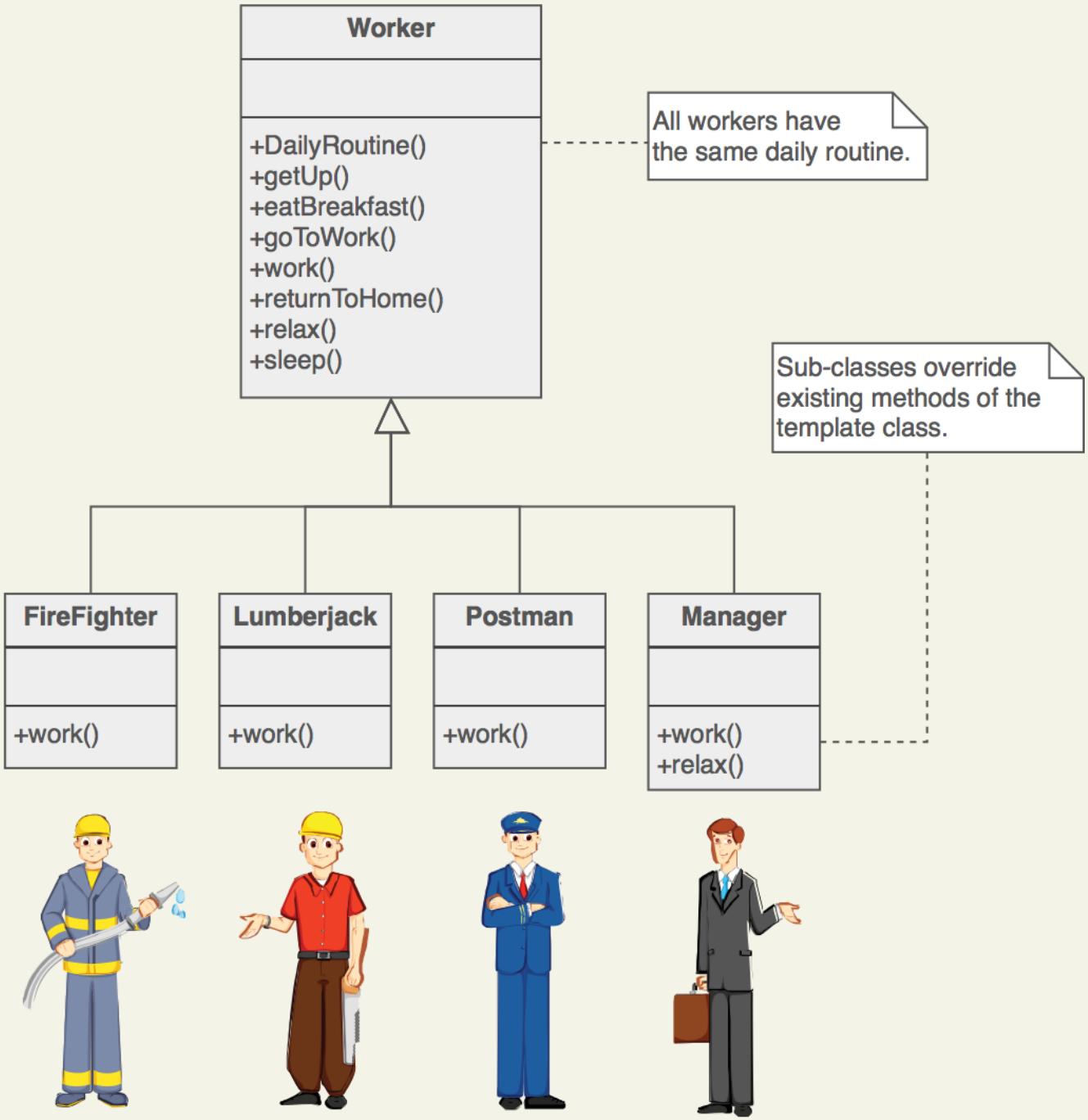
LISTING 14.3 Factory Method class for creating the adapters

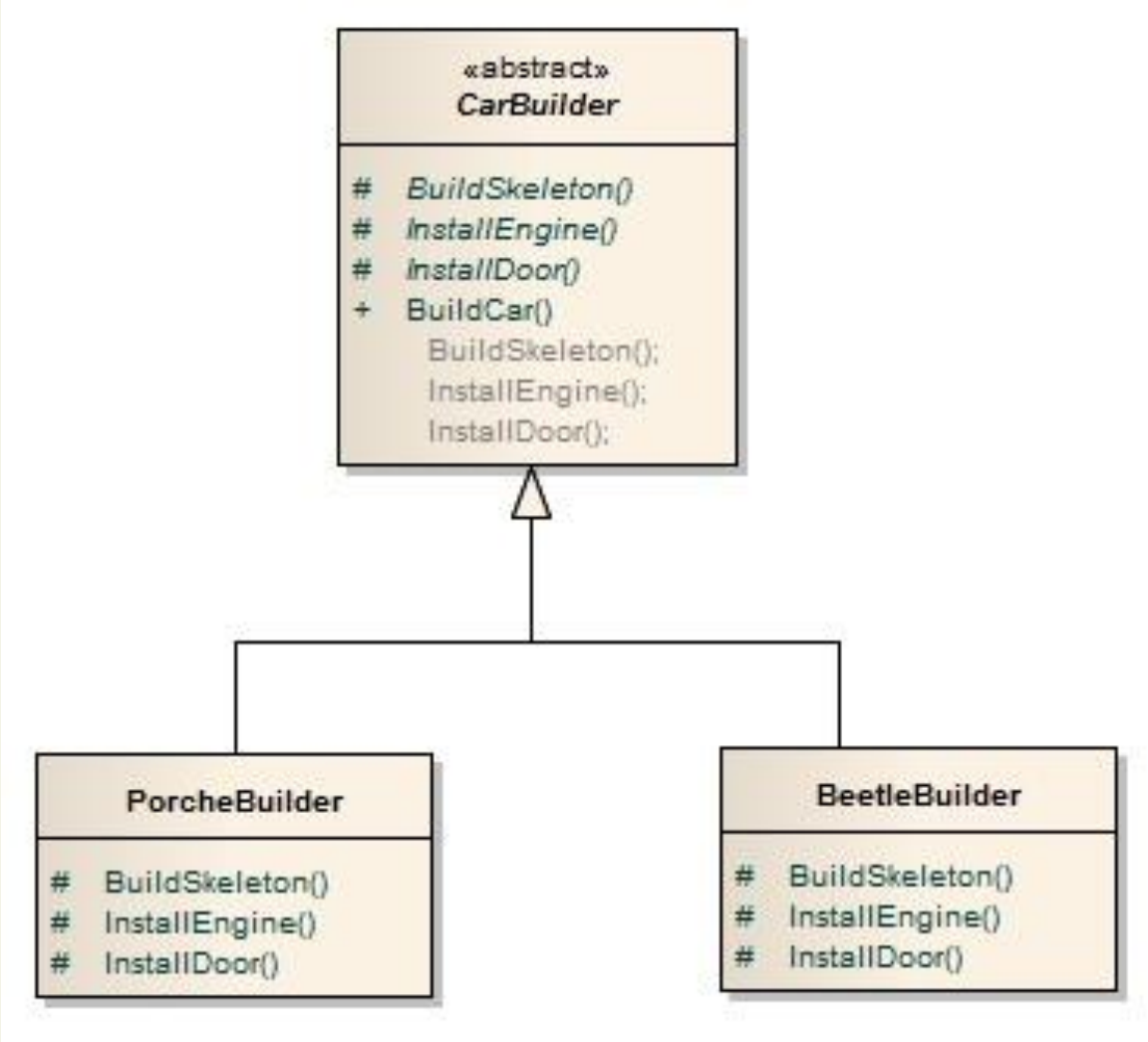
Template Method Pattern

In the Template Method pattern, one defines an algorithm in an abstract superclass and defers the algorithm steps that can vary to the subclasses.

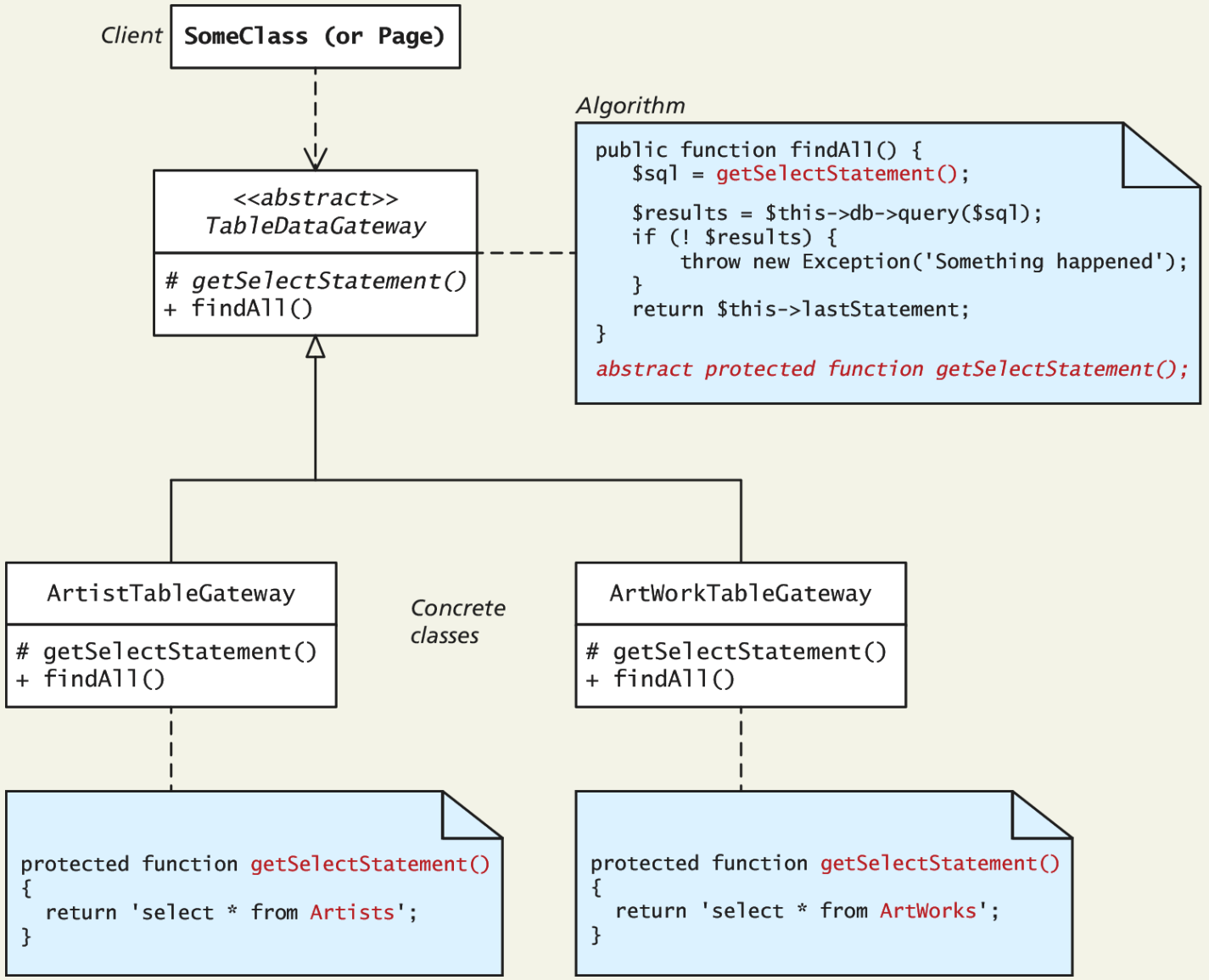
Like a shell with starter functions/classes

Game example


Worker
+DailyRoutine()
+getUp()
+eatBreakfast()
+goToWork()
+work()
+returnToHome()
+relax()
+sleep()
All workers have the same daily routine.
Sub-classes override existing methods of the template class.
FireFighter
+work()
Lumberjack
+work()
Postman
+work()
Manager
+work()
+relax()


«abstract»
CarBuilder
BuildSkeleton()
InstallEngine()
InstallDoor()
+ BuildCar()
BuildSkeleton();
InstallEngine();
InstallDoor();
PorcheBuilder
BuildSkeleton()
InstallEngine()
InstallDoor()
BeetleBuilder
BuildSkeleton()
InstallEngine()
InstallDoor()

Template Method Pattern



Template Method Pattern

Abstract Superclass

```
abstract class TableDataGateway
{
   ...
   //   The select statement for the table
   abstract protected function getSelectStatement();

   //  The name of the primary keys in the database
   abstract protected function getPrimaryKeyName();

   /*
     Returns all the records in the table
   */
   public function findAll()
   {
      $sql = $this->getSelectStatement();
      $results = $this->dbAdapter->fetchAsArray($sql);
      return $results;
   }
   /*
     Returns a single record indicated by the specified key field
   */
   public function findById($id)
   {
      $sql = $this->getSelectStatement();
      $sql .= ' WHERE ' . $this->getPrimaryKeyName() . '=:id';
      $result = $this->dbAdapter->fetchRow($sql, Array(':id' => $id));
      return $result;
   }
}
```

LISTING 14.4 Abstract super class for data access objects

Template Method Pattern

Example Subclasses

```
class ArtistTableGateway extends TableDataGateway
{
   ...
   protected function getSelectStatement()
   {
      return "SELECT ArtistID,FirstName,LastName,Nationality FROM
              Artists";
   }
   protected function getPrimaryKeyName() {
      return "AuthorID";
   }
}
class ArtWorkTableGateway extends TableDataGateway
{
   ...
   protected function getSelectStatement()
   {
      return "SELECT ArtWorkID,Title,Description,...FROM ArtWorks";
   }
   protected function getPrimaryKeyName() {
      return "ArtWorkID";
   }
}
```

LISTING 14.5 Example subclasses

Dependency Injection

reduce the number of dependencies

its purpose is to reduce the number of dependencies within a class,

injecting potential dependencies into a class rather than hard-coding them.

Configures from the outside vs. inside.

Pass the needed dependencies into the constructor

Consider the TableDataGateway class

The class needs an object that implements the DatabaseAdapterInterface in order to perform queries.

- One approach would be to provide a private data member in the TableDataGateway and instantiate the object in the constructor:

Dependency Injection

Example

```
abstract class TableDataGateway
{
   protected $dbAdapter;

   public function __construct($dbAdapter)
   {
      if (is_null($dbAdapter) )
         throw new Exception("Database adapter is null");

      $this->dbAdapter = $dbAdapter;
   }
   ...
}
```

LISTING 14.6 Dependency Injection example

```
$connect = array(DBCONNECTION, DBUSER, DBPASS);
$dbAdapter = DatabaseAdapterFactory::create(ADAPTERTYPE,$connect);
$gate = new ArtistTableGateway($dbAdapter);
```

Section 4 of 5

DATA AND DOMAIN PATTERNS

Enterprise Patterns

Enterprise patterns - provide best practices for the common type of big-picture architectural problems

Table Data Gateway Pattern

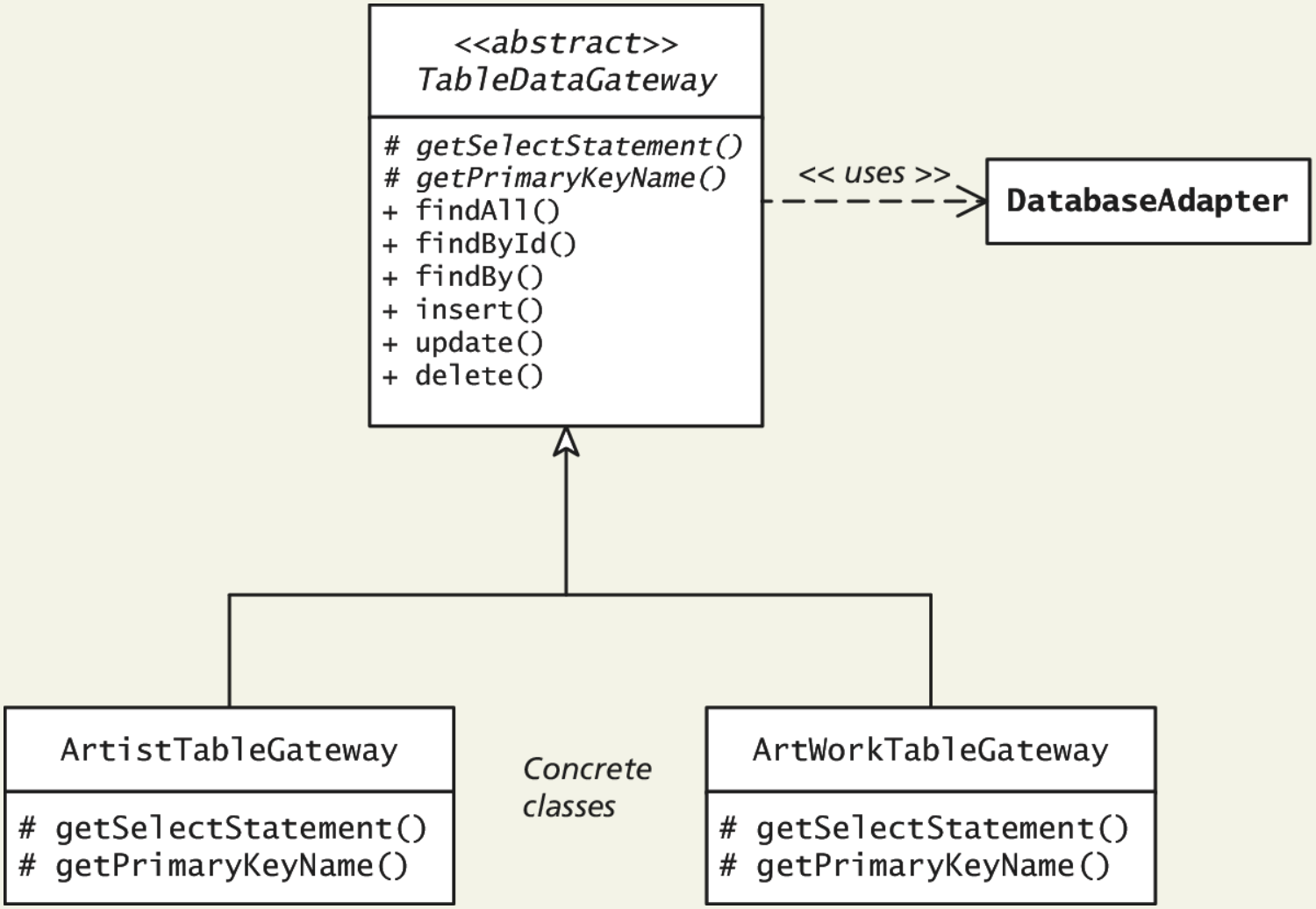
Data access Object

A **gateway** is simply an object that encapsulates access to some external resource.

Thus a table data gateway provides CRUD access to a database table (or perhaps joined tables).

Table Data Gateway Pattern

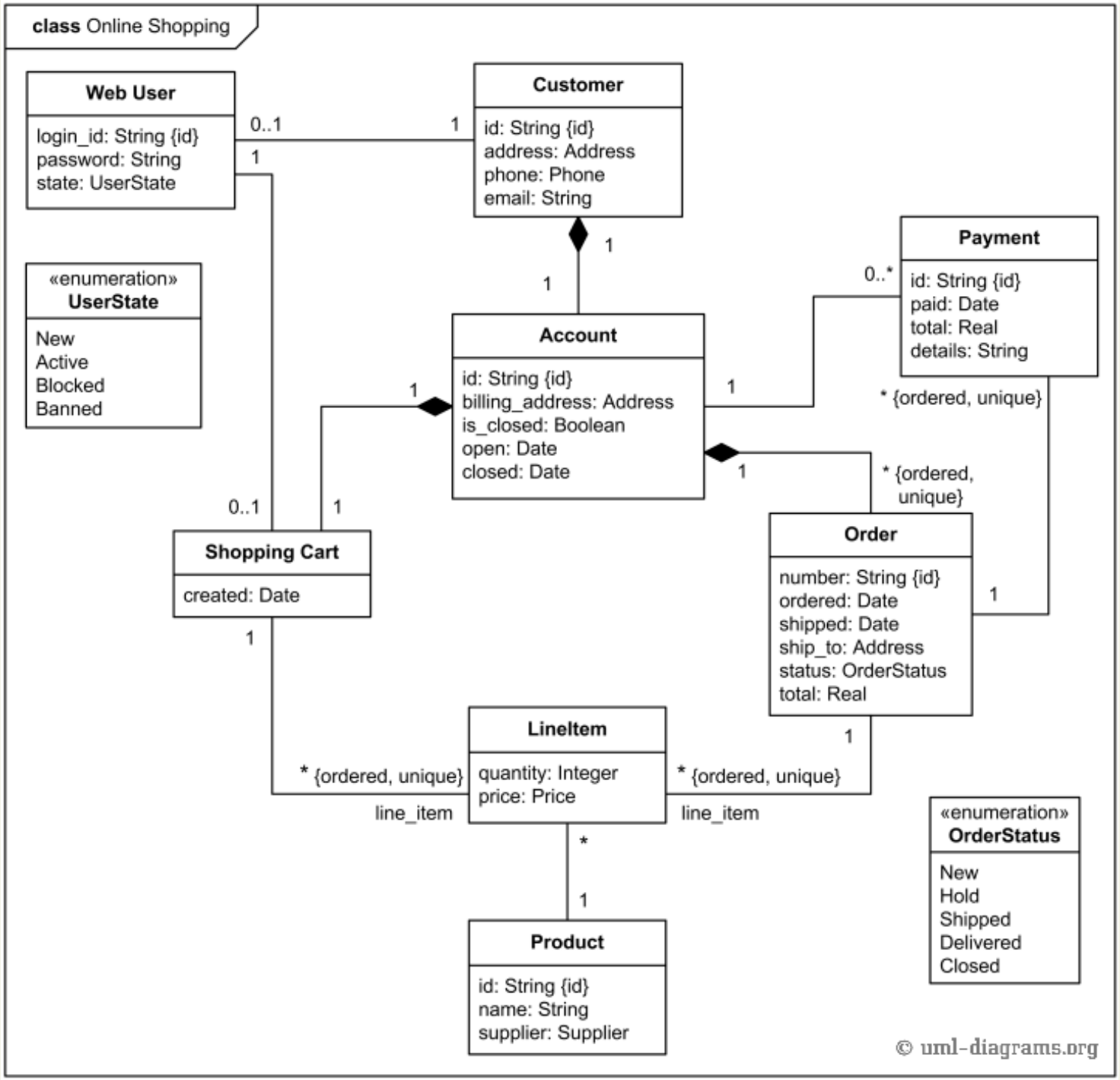
In UML



Domain Model Pattern

In the **Domain Model pattern** , the developer implements an **object model**:

- A variety of related classes that represent objects in the problem domain of the application.
- The classes within a domain model will have both data and behavior
- Natural location for implementing business rules.
- It can add precision and focus to discussion
- Similar to database schema


class Online Shopping
Web User
login_id: String {id}
password: String
state: UserState
0..1
1
Customer
id: String {id}
address: Address
phone: Phone
email: String
1
1
«enumeration»
UserState
New
Active
Blocked
Banned
Payment
id: String {id}
paid: Date
total: Real
details: String
0..*
Account
id: String {id}
billing_address: Address
is_closed: Boolean
open: Date
closed: Date
1
1
* {ordered, unique}
1
* {ordered, unique}
0..1
1
Shopping Cart
created: Date
1
Order
number: String {id}
ordered: Date
shipped: Date
ship_to: Address
status: OrderStatus
total: Real
1
1
LineItem
quantity: Integer
price: Price
* {ordered, unique}
line_item
* {ordered, unique}
line_item
*
1
Product
id: String {id}
name: String
supplier: Supplier
«enumeration»
OrderStatus
New
Hold
Shipped
Delivered
Closed
© uml-diagrams.org

Getters and Setters

In Domain Objects

Creating the properties along with their getters and setters for all the domain objects in a model can be very tedious.

PHP does provide its own type of shortcut via the __get() and __set() magic methods

The __get() method is called when a client of a class tries to access a property that is not accessible.

Magic occurs with the idea of Variable variables that PHP allows

“ $this-> “ is used to refer to an instance of an object inside on of the object's methods.

Getters and Setters

Magic

We could replace *all* of the property getters in the next slide with the following magic method:

```
public function __get($name) {
    if ( isset($this->$name) ) {
            return $this->$name;
    }
    return null;
}
```

```
class Artist
{
   // properties for the class
   private $id;
   private $firstName;
   private $lastName;
   private $nationality;
   private $yearOfBirth;
   private $yearOfDeath;

   // example getter and setter with validation
   public function getLastName() {
      return $this->lastName;
   }
   public function setLastName($value) {
      if (!is_string($value) || strlen($value) < 2 ||
          strlen($value) > 255) {
         throw new InvalidArgumentException("The last name is
                                             invalid.");
      }
      $this->lastName = $value;
   }
   // etc. ... getters and setters for other five properties

   // other behaviors
   public function getFullName($commaDelimited)  {
      if ($commaDelimited)
         return $this->lastName . ', ' . $this->firstName;
      else
         return $this->firstName . ' ' . $this->lastName;
   }
   public function getLifeSpan() {
      return $this->yearOfDeath - $this->yearOfBirth;
   }
}
```

LISTING 14.7 Example of simple domain object

Variable Variables

Magic Indeed

For instance

- if $name contains the string 'yearOfBirth'
- then $this->$name == $this->yearOfBirth.

__Set()

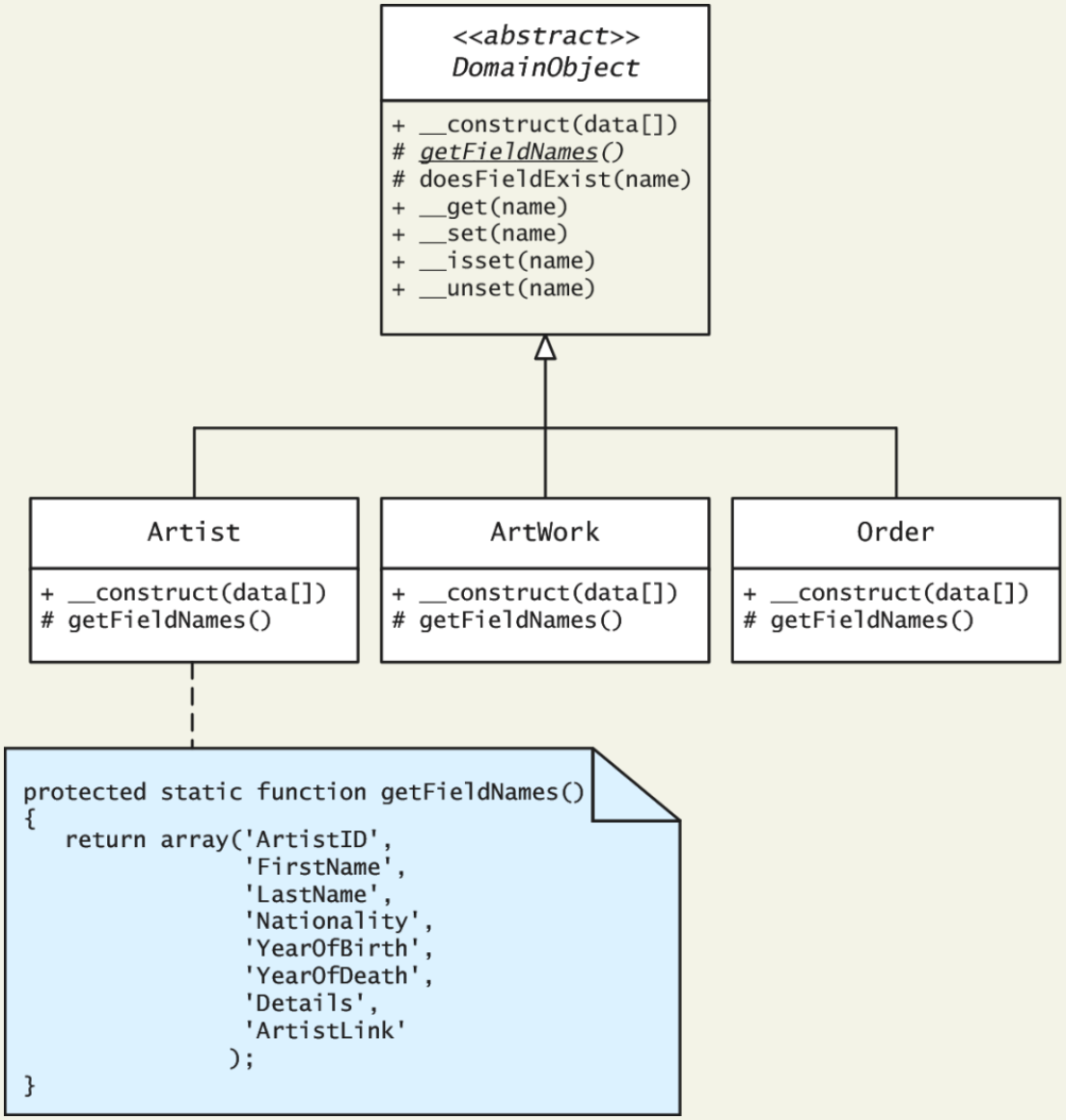
Example usage

```
class DomainObject {

  ...

  public function  __set($name, $value) {
     $mutator = 'set' . ucfirst($name);
     // if mutator method is defined than call it
     if (method_exists($this, $mutator) &&
         is_callable(array($this, $mutator)))    {
        $this->$mutator($value);
     }
     else {
        $this->$name = $value;
     }
  }
}
```

LISTING 14.8 Example __set() magic method

Example

Example Domain Model



Example

Example Domain Class

```
class Artist extends DomainObject
{
   static function getFieldNames() {
      return array('ArtistID','FirstName','LastName','Nationality',
        'YearOfBirth', 'YearOfDeath','Details','ArtistLink');
   }

   public function __construct(array $data)  {
      parent::__construct($data);
   }

   // implement any setters that need input checking/validation

   public function setLastName($value) {
      if (!is_string($value) || strlen($value) < 2 ||
        strlen($value) > 255) {
         throw new InvalidArgumentException("The last name is
                                             invalid.");
      }
      $this->lastName = $value;
   }

   // implement any other behavior needed by this domain object
}
```

LISTING 14.9 Example domain class

Domain Object and Gateway

Retrieving and Saving

```
// use artist gateway to retrieve a specific artist
$gate = new ArtistTableGateway($dbAdapter);
$artist = $gate->findByKey($id);
echo $artist->LastName . ', ' . $artist->FirstName;
...
// make a change to domain object
$artist->LastName = 'Picasso';
// then use gateway to save it
$gate->update($artist);
```

LISTING 14.10 Retrieving and saving data using a domain object and a gateway

Active Record Pattern

Interface with the database

You may be wondering what class would have the responsibility of populating the domain objects from the database data or of writing the data within the domain object back out to the database.

Keep in mind the idea of the Activation Record in general programming, the temporary area in RAM.

It seems this relates to that but with full database data and methods.

Active Record Pattern

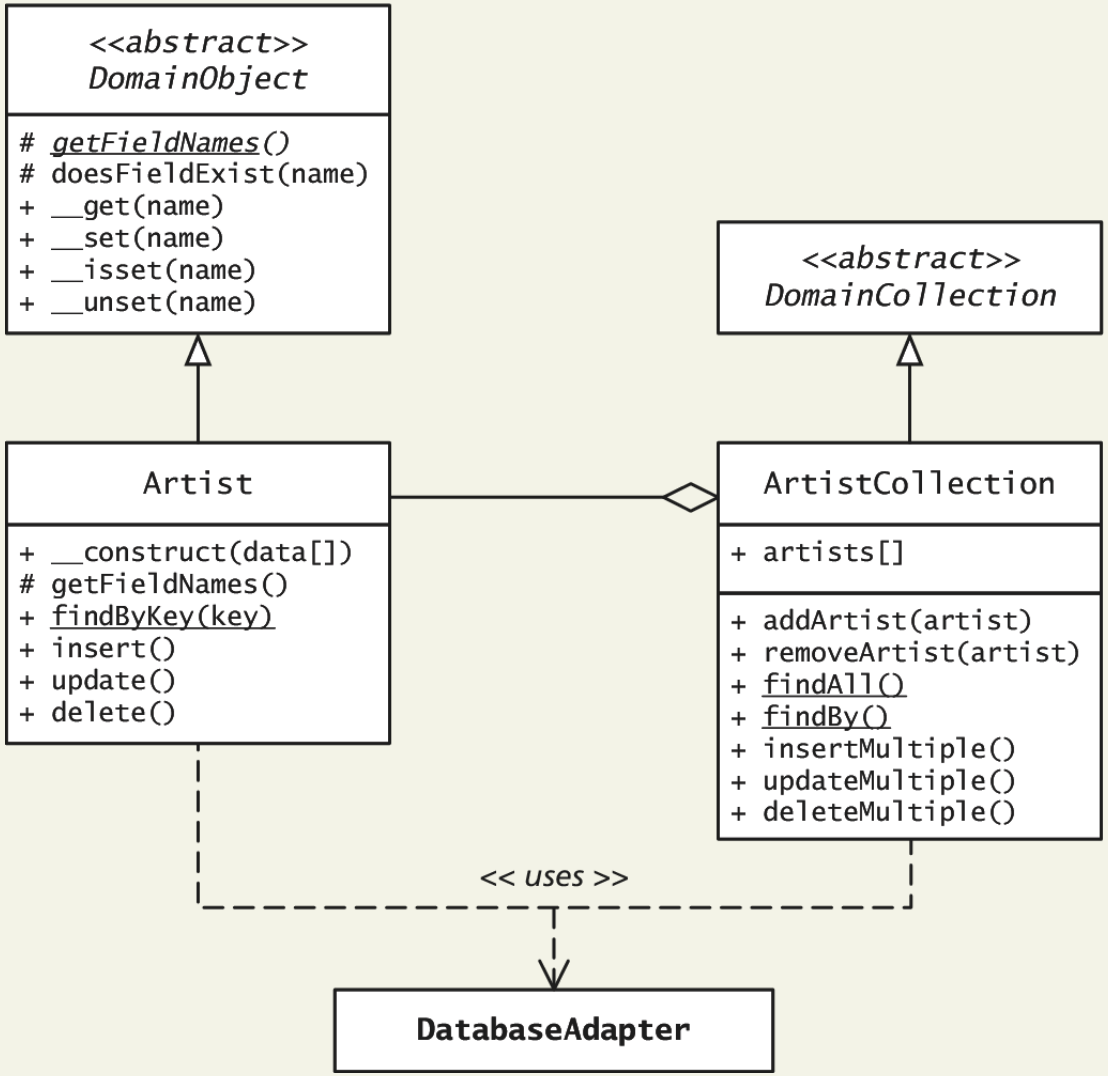
Interface with the database

```
// use static method of Artist class to find a specific artist
$artist = Artist::findByKey($id);
echo $artist->LastName . ', ' . $artist->FirstName;
...
// make a change to domain object
$artist->LastName = 'Picasso';
// then tell domain object to update itself
$artist->update();
```

LISTING 14.11 Retrieving and saving data using active record pattern

Active Record Pattern

Retrieving and Saving



Section 5 of 5

PRESENTATION PATTERNS

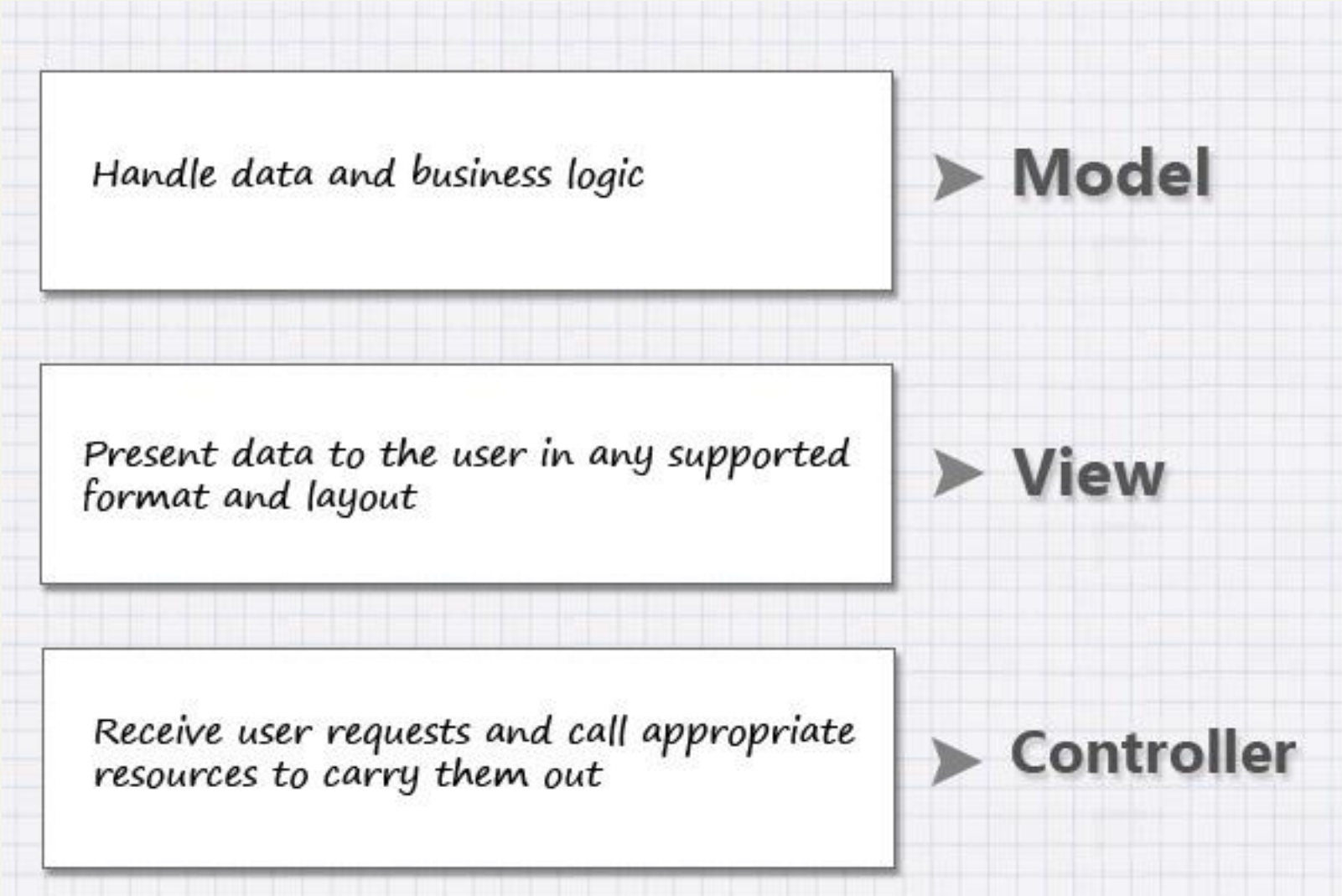
Model View Controller

MVC

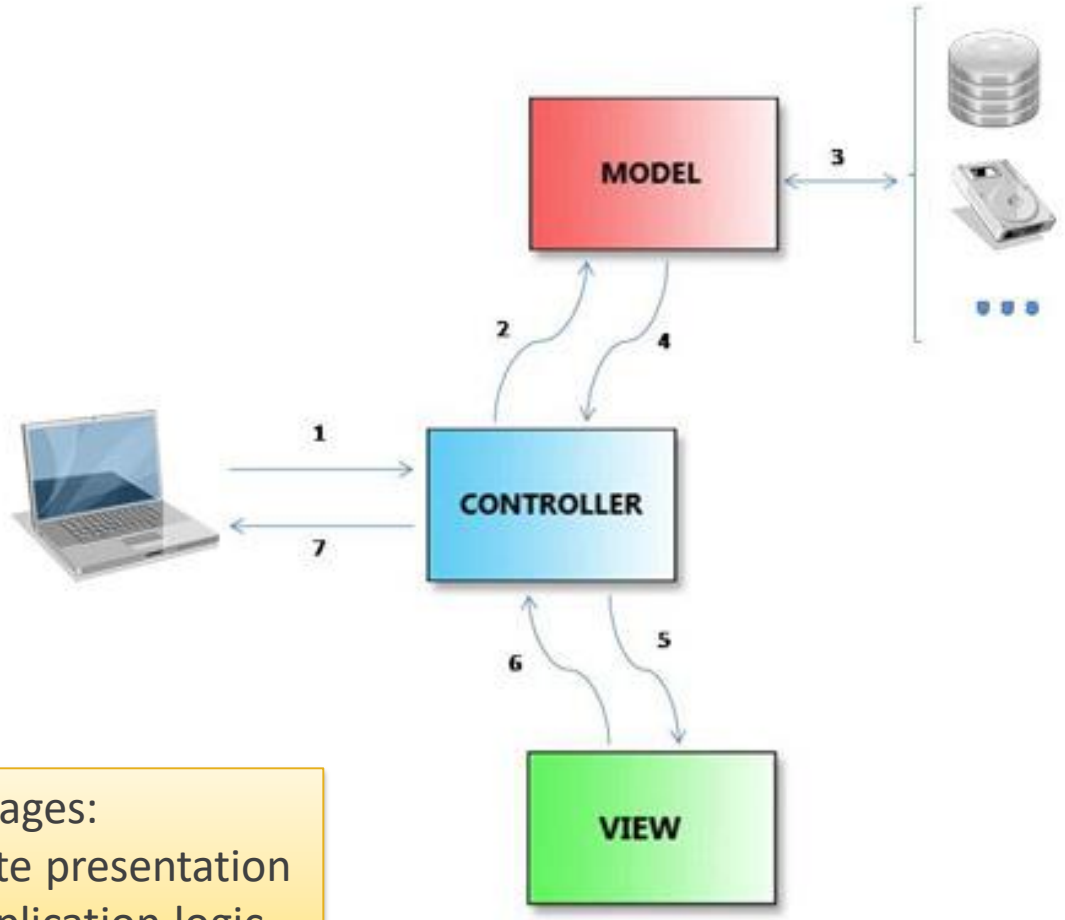
The **Model-View-Controller (MVC) pattern** actually predates the whole pattern movement.

- The **model** represents the data of the application
- The **view** represents the display aspects of the user interface.
- The **controller** acts as the “brains” of the application and coordinates activities between the view and the model.
- Describes a way to structure an application, its responsibilities and interactions for each part
- How about the WWW ?

Model View Controller for Web



Model View Controller example

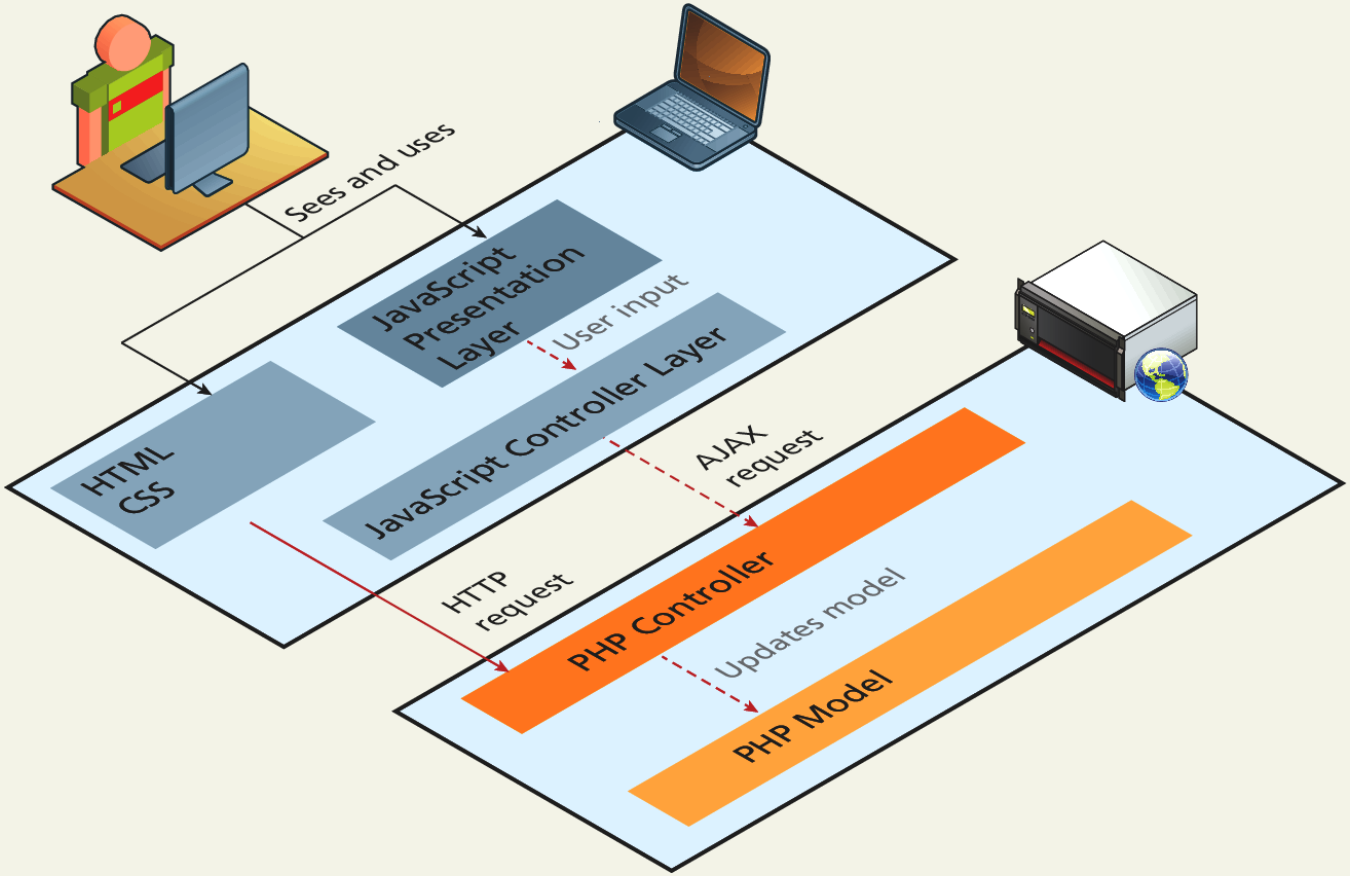


Advantages:
Separate presentation and application logic

On-line book store

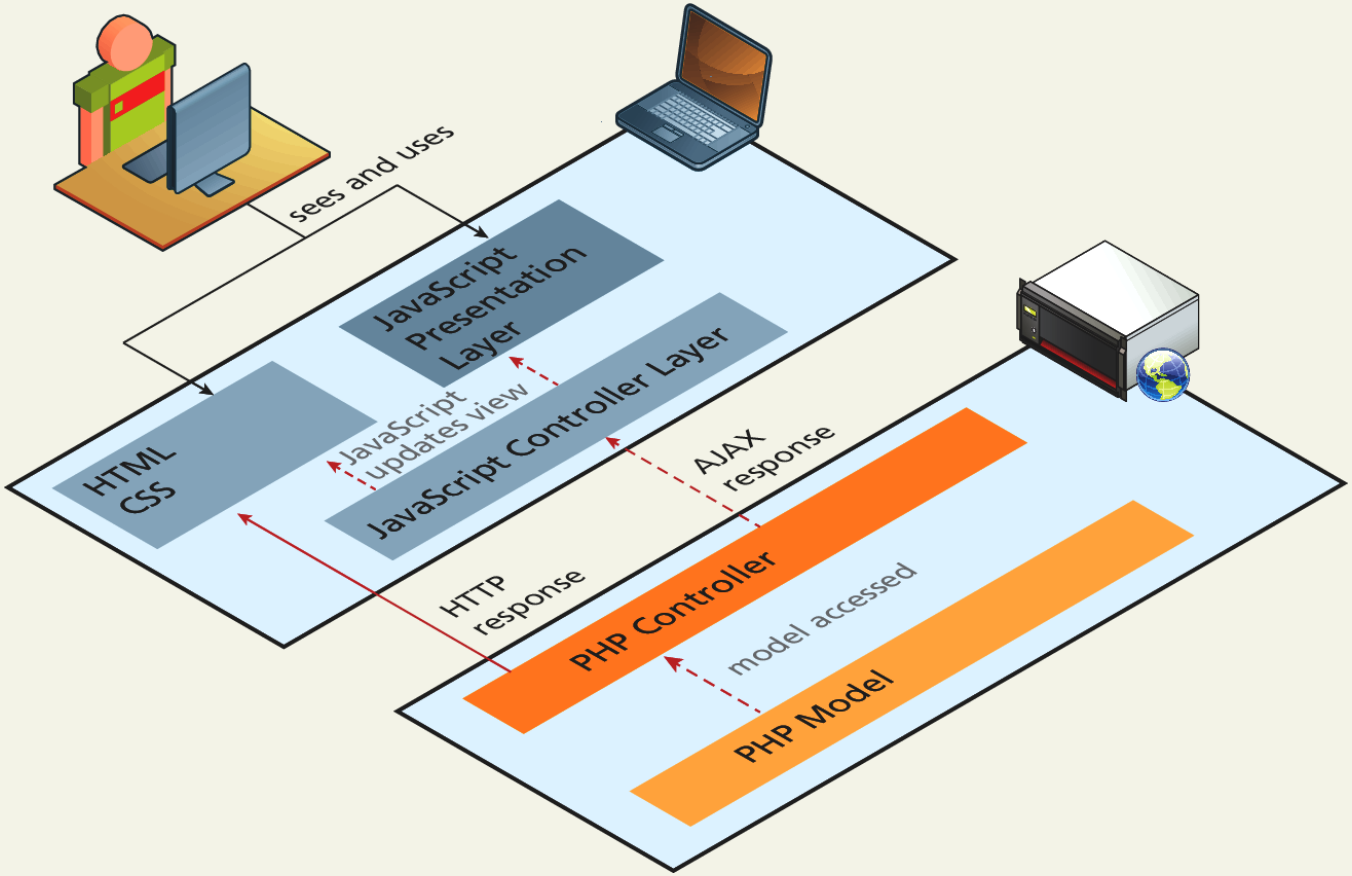
Model View Controller

MVC split between client and browser



Model View Controller

MVC split between client and browser (illustrated response)



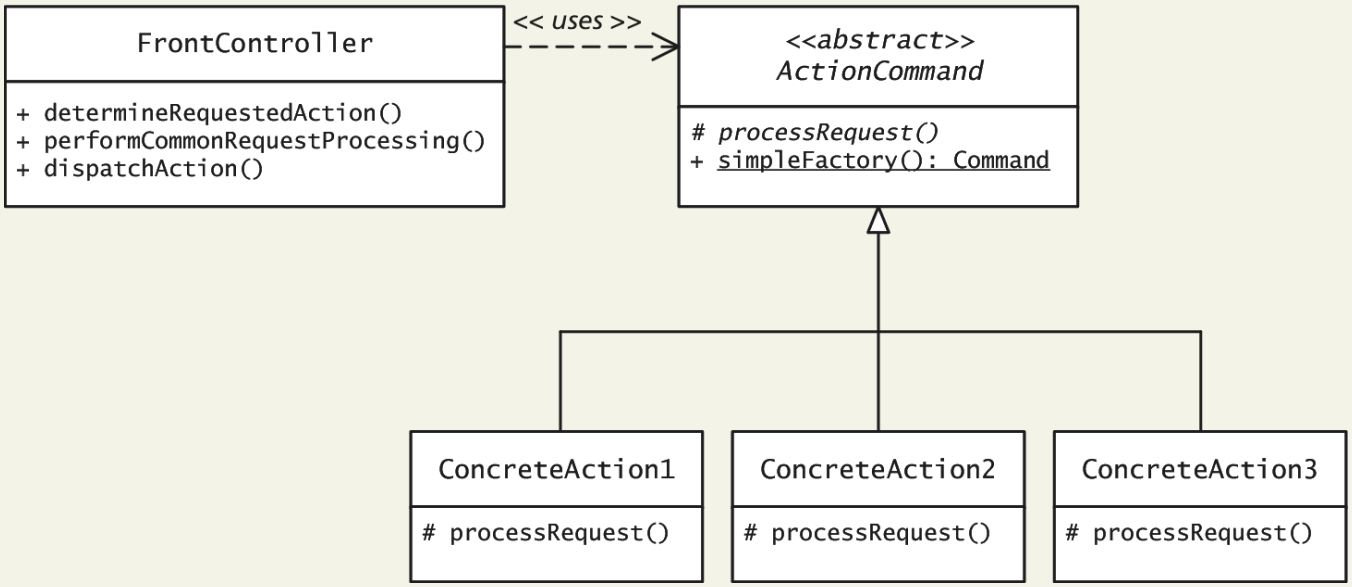
Front Controller Pattern

The Front Controller pattern consolidates all request handling into a single-handler class.

The rationale for the front controller is that in more complex websites every request requires similar types of processing.

- One approach to this standardized behavior is to provide this functionality to each page via common include files.
- A more object-oriented approach is to use a front controller, in which one (or a small number) script or class is responsible for handling every incoming request and then delegating the rest of the handling to the appropriate handler.

Front Controller Pattern



What You've Learned

1. Real World Web Software Design
2. Principle of Layering
3. Design Patterns in Web Context
4. Data and Domain Patterns
5. Presentation Patterns