OBJECT / RELATIONAL MAPPING:

OBJECT ORIENTED DATABASE PROGRAMMING IN PHP

John Valance

Division 1 Systems

<div1>

johnv@div1sys.com

About John Valance

- Independent Consultant for 15 years
- Founder and CTO of Division 1 Systems
  - Helping IBM shops develop web applications and related skills
  - Extended team of 150+ technical people
  - Web and mobile systems development, design, project management
  - Training, mentoring, consultation and coding
- 30+ years IBM midrange experience (S/38 thru IBM i)
- 15+ years of web development experience
- Frequent presenter on web development topics
- Relationship with Zend Technologies
  - Teach Intro to PHP for RPG programmers
  - Zend Certified Engineer
  - Zend Reseller

We Will Cover:

- **Object Relational Mapping(!) aka ORM**
- Object Relational (Impedance) Mismatch
- OOP vs. RDBMS – what and why
- Object Oriented Design Patterns
- Martin Fowler Book:
  - Patterns of Enterprise Application Architecture
- Common RDB Patterns
  - Simple to Complex
- PHP Code Samples
  - Illustrating the Patterns

Object Relational Mapping (ORM)

- Objects and Databases:
  - Both deal with *Complex Data Structures*
  - Objects = data exists in memory during program execution
  - RDBMS = persist (i.e. store) data on disk for use by many applications
- Object Classes and Databases differ in numerous ways
  - Data types, Relationships, etc… (next slide)
- Need a way to **Map** between **Objects** and **Relational** DBMS representations of data
- Object / Relational Mapping = ORM
  - **That's what we are covering in this presentation!** **(using PHP)**

OOP vs. RDBMS - Similarities

	OOP	RDBMS
Data Structures defined by	Class	Table
Instance of data structure	Objects	Row
Field of Data Structure	Property	Column
State Operations	Class methods : create(), find(), save(), delete()	DML : insert, select, update, delete
Relationships between entities	Pointers, Nested objects, Inheritance	Data values (i.e. foreign key columns)

- Both manage changes of state to a data instance (CRUD)

ORM: Map SP_CUST table to OO Class

RDBMS: SP_CUST Table Layout

ZENDSVR6/SP_CUST

Name		Type	Len	Prec	Scale	Description
CUST_ID	1	Numeric		5	0	CUSTOMER ID
COMPANY	2	Char	30			COMPANY NAME
FIRSTNAME	3	Char	20			CONTACT FIRST NAME
LASTNAME	4	Char	20			CONTACT LAST NAME
CIVIL	5	Char	1			
ADDRESS	6	Char	30			ADDRESS
ADDR2	7	Char	30			
CITY	8	Char	15			
STATE	9	Char	20			STATE
ZIP	10	Char	10			ZIP
COUNTRY	11	Char	20			COUNTRY
PHONE	12	Char	15			PHONE
FAX	13	Char	15			FAX

OO PHP: Customer Class

Looks pretty simple, eh??

Customer.php

```
1  <?php
2
3  class Customer {
4      private $CUST_ID;
5      private $COMPANY;
6      private $FIRSTNAME
7      private $LASTNAME;
8      private $CIVIL;
9      private $ADDRESS;
10     private $ADDR2;
11     private $CITY;
12     private $STATE;
13     private $ZIP;
14     private $COUNTRY;
15     private $PHONE;
16     private $FAX;
17
```

Outline

- use statements
- Customer
  - load($data)
  - getFormattedAddress($html
  - getMailingLabel($html=false
  - getCUST_ID()
  - getCOMPANY()
  - getFIRSTNAME()
  - getLASTNAME()
  - getCIVIL()
  - getADDRESS()
  - getADDR2()
  - getCITY()
  - getSTATE()
  - getZIP()
  - getCOUNTRY()
  - getPHONE()
  - getFAX()
  - setCUST_ID($CUST_ID)
  - setCOMPANY($COMPANY)
  - setFIRSTNAME($FIRSTNAME

Object-Relational Impedance Mismatch

- Term used to refer to the problems encountered when mapping RDBMS schema into an Object Oriented programming model

- Borrows the term "Impedance Mismatch" from electrical engineering

"Impedance is the opposition by a system to the flow of energy from a source." – Wikipedia
https://en.wikipedia.org/wiki/Impedance_matching#Theory

*The **object-relational impedance mismatch** is a set of conceptual and technical difficulties that are often encountered when a relational database management system (RDBMS) is being served by an application program (or multiple application programs) written in an object-oriented programming language or style, particularly because objects or class definitions must be mapped to database tables defined by relational schemata.*
(https://en.wikipedia.org/wiki/Object-relational_impedance_mismatch)

OOP vs. RDBMS - Differences

	OOP	RDBMS
Where data exists	In Memory	On disk
Entity relationships	Hierarchical / Network model via nested objects/inheritance	Network of related tables (via foreign key data)
Implementation	Physical Pointers	Logical Pointers (foreign key columns) and Indexes
Based on	Software Engineering principles	Mathematical principles (set theory)
Programming paradigm	Object Oriented Instance-based operations Methods change object state	SQL / Declarative Set-based operations DML commands change state
Data Types	Language dependent / Class-based	DBMS dependent
Business rules	Encapsulation (code tied to data in Class definition)	Constraints / Triggers / SQL

What's the Solution

- Highly debatable question!
- The short answer: "It depends…"
  - No one-size-fits-all solution
- We will look at the OO perspective, and discuss Patterns
  - This can get pretty deep… 😱
  - We will go only so far. 😊
- How deep you go depends on
  - Where you're coming from
  - Where you want to get to
- Best to start simply, based on your strengths
  - Typical IBM i shop has lots of RDBMS experience

The Vietnam of Computer Science

For some people, ORM has no solution, like this bloody conflict:

Famous article by Ted Neward:
The Vietnam of Computer Science (2006)
http://blogs.tedneward.com/post/the-vietnam-of-computer-science/

- Neward compares the ongoing struggle to reconcile the differences between Relational Database design and Object Oriented design, to the Vietnam War, and its no-win outcome.
  - He argues that ORM is a quagmire with no good solution
  - The further you get into it, the harder it is to get out, and the more you regret going in in the first place
- He summarizes both the issues of the Vietnam War, and the issues of ORM

ORM Mapping Problems *

- Object-to-Table Mapping problem
  - Inheritance is very natural for modeling objects, but not supported in RDBMS
- Schema Ownership conflict
  - DBA vs. Developer
- Dual-Schema problem
  - Metadata exists in both RDBMS and in OO code base
- Entity Identity Issues
  - After data is loaded into object, how to ensure no other updates occur in the database
- Data Retrieval problem
  - OOP does not support SQL natively.
  - OO-based query languages or classes/frameworks are awkward

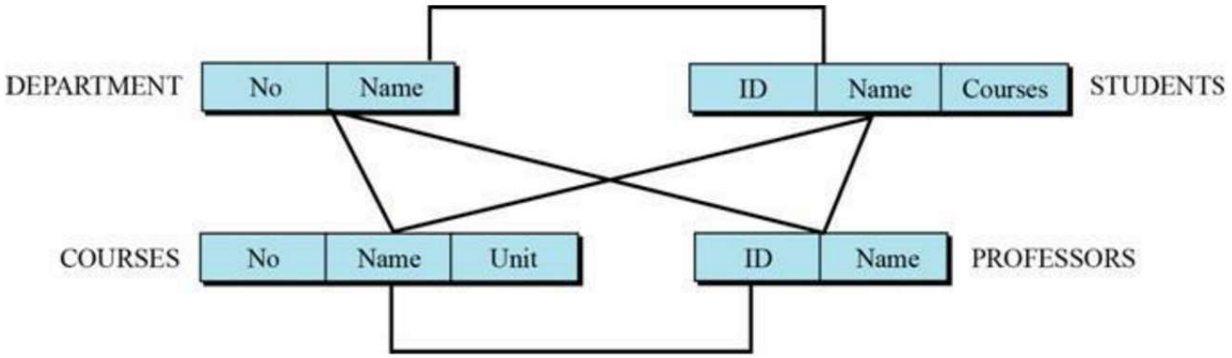
* Based on points made by Ted Neward, in his article "*The Vietnam of Computer Science*"
http://blogs.tedneward.com/post/the-vietnam-of-computer-science/

ORM Mapping Problems

Partial Objects and the Load-Time Paradox *

- Typically objects form a network of interconnections in any system

Example:



- Instantiating one object can have a cascading effect.
- To be a complete data model, all possible relations should be populated at the same time, right?? *Hmmm…*
  - *Can you say performance problems?*
- Need to use the "Lazy-Load" pattern *(your mileage may vary)*

* Ted Neward, "*The Vietnam of Computer Science*"

Is There a Winner in this Conflict?

Are we comparing Apples to Oranges?

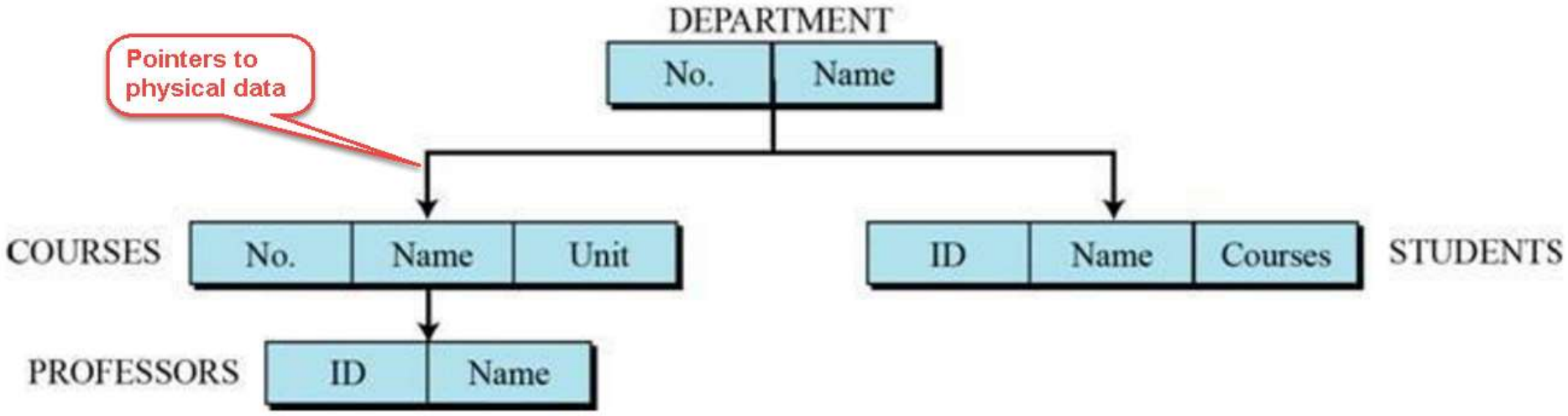
If only we could just concern ourselves with one or the other!

DATABASES

Why do we need the DBMS side of the equation

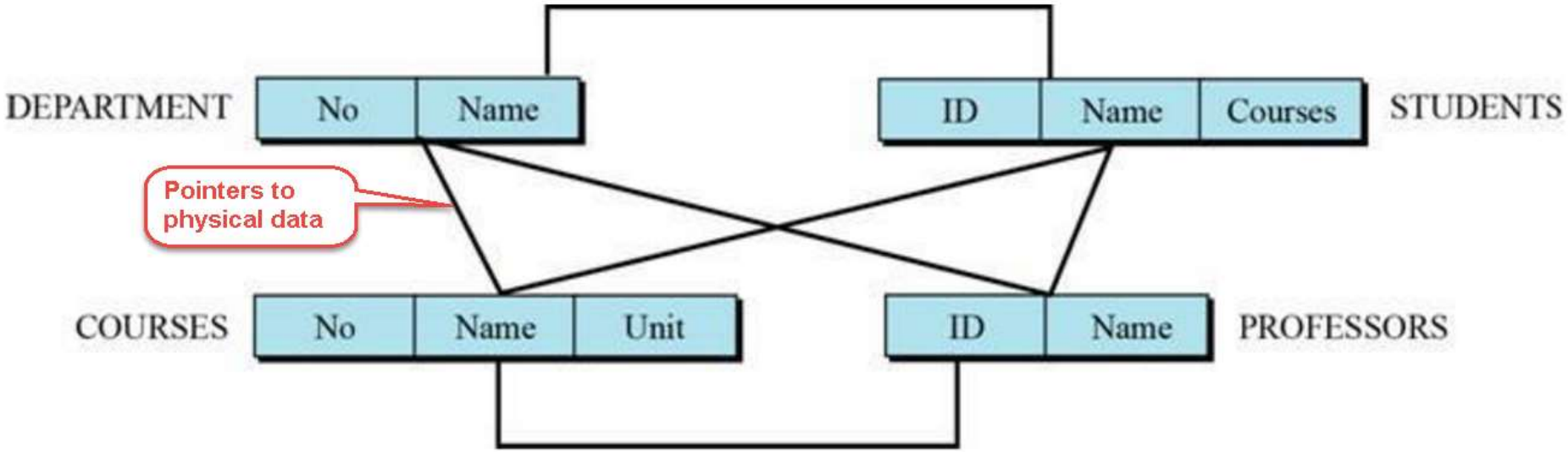
DBMS history: 1970s - Hierarchical Model

- Hierarchical relations (one-to-many)
- Relationships implemented via pointers to physical data!
- Inverting relations (i.e. ORDER BY) not possible without creating duplication of data and a new structure.



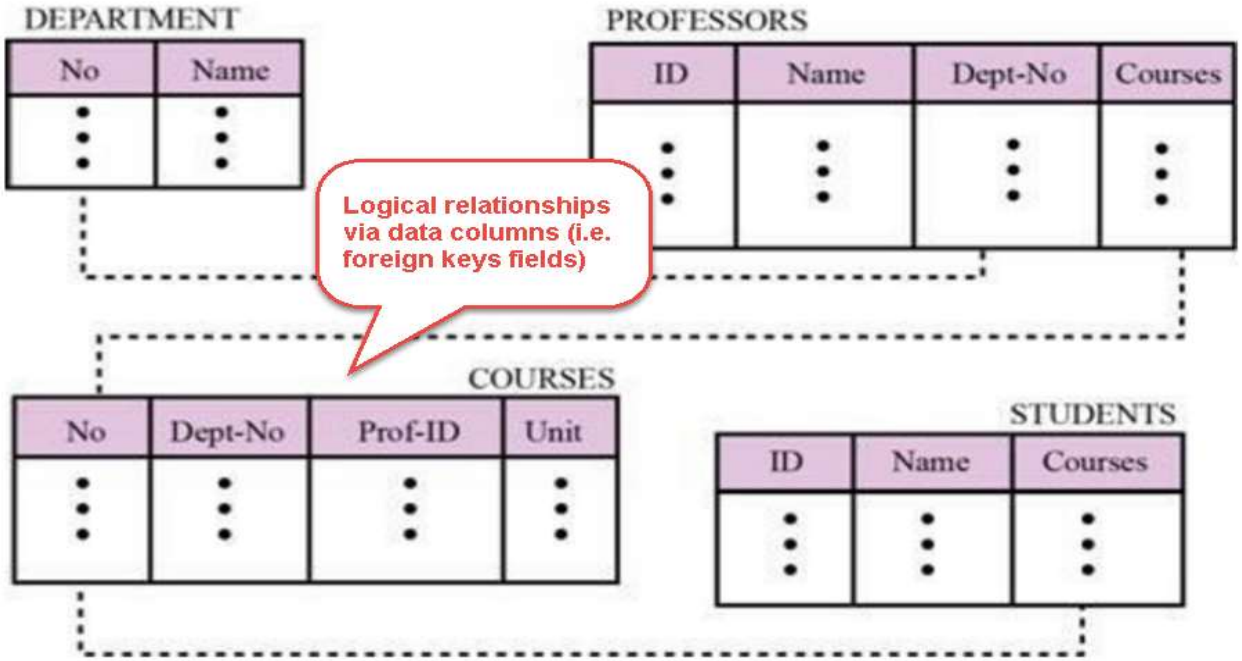
DBMS history: 1960s - Network Model

- All possible relationships (many-to-many)
- Relationships implemented via pointers to physical data!
- Similar problems to Hierarchical Model



DBMS history: 1970s - Relational Model

- Tables of rows/columns, related via data values (foreign key fields).
- Logical abstract relationships - **No physical pointers!**



Ascendance of RDBMS

- Abstraction of the logical vs. physical models
  - Just the kind of thing we love about OO!
- Based on Mathematical Set Theory
  - Relational Calculus == SQL
  - Powerful, ad-hoc, set-based operations possible
- Relationships established by data values (foreign keys)
  - Not using physical pointers
  - Indexing tables required for performance
- Extremely flexible
  - Tabular, Hierarchical and Network models all possible using Views, SQL and Indexes
- At first, this complex abstraction did not perform well
  - Moore's Law took care of this before long
- Currently **ubiquitous in enterprise environments**

Why Use RDBMS

- Extremely Flexible Model
  - Abstraction of logical model from physical implementation.
- Provides data persistence (objects do not)
- Legacy applications (RDBMS is ubiquitous)
- High Performance – transactional systems
- SQL - Simple, standardized syntax
- Other advanced features
  - Triggers, User defined functions, Stored procedures, Views, Concurrency, Transaction Management, Security, etc.
- Tooling / Resources
  - Well established, widely used skills and technology

OBJECTS

Why do we need the Object Oriented side of the equation

Why Use Objects

OO comes from the experiences of software application development

- Further abstraction at application level
  - Object modeling allows focus on business logic (i.e. Domain Logic)
- Modularity / Encapsulation
  - Self-contained collections of code and data
- Extensibility / Reusability
  - Inheritance, Plug-ins
- Reduce Maintenance Costs
  - Single point of maintenance
- Tooling / Resources
  - Well established, widely used skills and technology
- Design is paramount
  - Encourages planning ahead and thinking of consequences
- Scaling
  - OO more important in large, complex systems
- Testing
  - Automated testing possible (with certain restrictions)

OO Design Patterns

- General, reusable solution to a commonly occurring problem
  - NOT a finished design that can be transformed directly into code.
  - A description or template of best practices for solving a common problem
  - Can be used in many different situations.
- An attempt to organize the complexity of software architecture options
- Tries to establish a common language around which developers can discuss the options for system architecture
  - e.g. “Pattern Catalogs”
- Somewhere between the concept of object-oriented programming and a concrete algorithm.

Patterns for the RPG programmer

- Problem: Display a list of database records
  Design Pattern: **Subfile**
- Problem: Print a Report with Subtotals
  Design Pattern: **Level Breaks**
- Problem: Ensure DB transaction integrity
  Design Pattern: **Commitment Control**
- Problem: Process Multiple Requests for System Resources
  Design Pattern: **Job Queue**
- Problem: Allow programs to run in different environments without recoding
  Design Pattern: **Library Lists**

Design Patterns - HISTORY

- 1970s: Started as architectural concept
  - Chistopher Alexander – American architect, late 1970s
- 1980s: Similar concepts in software architecture
- 1994: publication of the Gang of Four book:
  - Design Patterns: Elements of Reusable Object-Oriented Software
    by: Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides (aka "*The Gang of Four*")
    - The classic, seminal book on object oriented design patterns
    - Includes coding examples in C++ and Smalltalk
- Late 1990s: OO design common-place. Rise of OO patterns
- 2002: publication of Patterns of Enterprise Application Architecture, by Martin Fowler

Martin Fowler book

Patterns of Enterprise Application Architecture

- by Martin Fowler
- Addison-Wesley Professional; 1 edition (November 15, 2002)

- **Seminal book on OO Patterns**, largely related to ORM
- Describes classic patterns for dealing with enterprise applications
- Basis for the patterns detailed in this presentation
- Clearly discusses the issues faced by ORM, describing problems and solutions with code examples.
  - Code examples in Java and C#

Some Patterns Described by Fowler

- Object Generation Patterns
  - Singleton
  - Factory

- ORM / Database Application Patterns
  - Table Data Gateway
  - Row Data Gateway
  - Active Record
  - Data Mapper
  - Metadata Mapper
  - Query Builder

Considerations for Selecting ORM Patterns

- Database Access
  - Should class contain database access functions (i.e, db2_connect)?
  - or should db2 access be factored into a separate class?
- Data Mapping
  - Should database fields be modeled as properties of a class?
    - This implies an object instance per table row
    - How do we map the database row to an object instance?
  - Should the class have no properties, just a collection of methods (operations) that can be performed on any row passed as parameter?
- Domain Logic
  - Domain logic ~= business logic
  - Should the class contain domain logic? data mapping? database access? All 3? Just 2 of these?

ORM / Database Patterns

- Table Data Gateway
  - Each instance represents a table as a whole
  - No properties that map data fields
- Row Data Gateway
  - Each instance represents a row in the table
  - Properties represent database fields
- Active Record
  - Like Row Data Gateway, but adds domain logic
- Data Mapper
  - Involves 2 separate objects:
    - Domain object, containing properties and methods relevant to object model
    - Mapper object, which maps data between domain object and database
  - Allows object model and database model to evolve separately
- MetaData Mapper
  - A mapper that can work with any domain object, using metadata from the database system catalog to perform generic mapping.
- Query Builder
  - An OO abstraction of SQL queries

Sample Application

- PHP code samples for each pattern provided
- Start with procedural code that accesses database
  - Perform various CRUD operations: Create, Read, Update, Delete
- We will then rewrite the application, using various ORM patterns
- We are mapping a very simple PERSON table:

```
create table jvalance.person (
    id integer not null
        generated always as identity,
    firstName char(30),
    lastName char(40),
    dateOfBirth date
);
```


Some Common Patterns

- Object Generation Patterns
  - Singleton
  - Factory
- ORM / Database Application Patterns
  - Table Data Gateway
  - Row Data Gateway
  - Active Record
  - Data Mapper
  - Metadata Mapper
  - Query Builder

Singleton pattern

- Ensures only one instance of an object can be created
- The same instance can be created and used by many other objects, without any parameter passing
- Useful for database connections, and other expensive, shareable resources

BAD – Creating multiple connections

- Connecting locally to database, more than once
- Very expensive operation

```
function getCustomers () {
    $conn = db2_connect ("*LOCAL", USER, PSWD, $conn_opts );
    db2_prepare($conn, 'select * from customer');
    // ...
}
function getItems () {
    $conn = db2_connect ("*LOCAL", USER, PSWD, $conn_opts );
    db2_prepare($conn, 'select * from items');
    // ...
}
```


BETTER – Pass connection parameter

- Connect to database once in mainline
- Pass connection where needed

```
$conn = db2_connect ("*LOCAL", USER, PSWD, $conn_opts );
getCustomers( $conn );
getItems( $conn );
//-------------------------------------------------------
function getCustomers ( $conn ) {
    db2_prepare($conn, 'select * from customer');
    // ...
}
function getItems ( $conn ) {
    db2_prepare($conn, 'select * from items');
    // ...
}
```

- But… never know how deep into object stack you'll be!
  - You may need to add connection parameter to every method in your system! 😱

MUCH BETTER – Singleton connection class

- Can call static method *getDb2Connection*(), wherever needed.
- Will only connect once. Always returns same connection resource.
- No parameter passing involved

```
require_once 'DB2Connection.php'; // singleton class

function getCustomers( ) {
    $conn = DB2Connection::getDB2Connection();
    db2_prepare($conn, 'select * from customer');
    // ...
}
function getItems( ) {
    $conn = DB2Connection::getDB2Connection();
    db2_prepare($conn, 'select * from items');
    // ...
}
```

Singleton relies on Static Class Members

Static methods and properties can be accessed without an object instance

```
class MyClass {
    private static $staticProperty;
    private $instanceProperty;

    public static function setStaticProperty( $staticValue ) {
        self::$staticProperty = $staticValue;
    }

    public function setInstanceProperty( $instanceValue ) {
        $this->instanceProperty = $instanceValue;
    }
}

MyClass::setStaticProperty('Look ma, no object instance!');

$myObject1 = new MyClass();
$myObject1->setInstanceProperty('This is one object instance.');

$myObject2 = new MyClass();
$myObject2->setInstanceProperty('This is a different object instance.');
```


Singleton – How it Works

- Private constructor
  - Constructor can only be called from within another class method.
  - Ensures that constructor will only be called once
- Private, static connection property
  - Stores the single db2 connection resource
  - Must be static, so that static `getDB2Connection()` can access it.
- Static `getDB2Connection()` method
  - Controls generation and retrieval of the single object instance
  - If connection is null,
    - set it with db2_connect(), and
    - return the new connection
  - If not null,
    - return the already active connection

Singleton connection class in PHP

```
class DB2Connection {
    private static $connectionResource;

    // Private constructor - cannot be called outside this class!
    private function __construct() {  }

    // Static method - can be called without object instance
    public static function getDB2Connection() {
        if ( empty(self::$connectionResource) ) {
            self::$connectionResource = db2_connect(DB, USER, PSWD);
        }
        return self::$connectionResource;
    }
}

$db2Conn = DB2Connection::getDB2Connection();
db2_exec($db2Conn, 'select * from items');
```

Use `::` to reference static methods and properties

`::` is called the Scope Resolution operator, or double colon
(or Paamayim Nekudotayim, which means double colon in Hebrew!)

Some Common Patterns

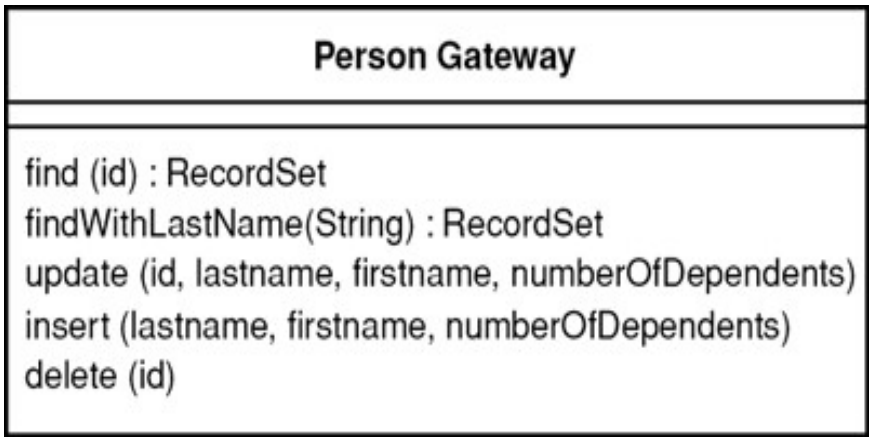
- Object Generation Patterns
  - Singleton
  - Factory

- ORM / Database Application Patterns
  - Table Data Gateway
  - Row Data Gateway
  - Active Record
  - Data Mapper
  - Metadata Mapper
  - Query Builder

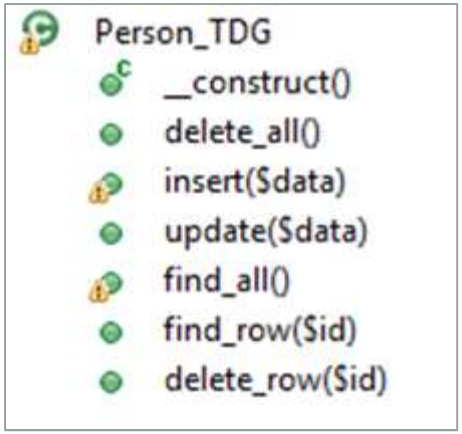
Table Data Gateway (TDG)

An object that acts as a Gateway to a database table. One instance handles all the rows in the table.

- Usually consisting of several find methods to get data from the database and update, insert, and delete methods.
- Each method maps the input parameters into a SQL call and executes the SQL against a database connection.
- No state (no properties) just for data transfer
- Find methods can return record sets (i.e., arrays of objects)
- aka – Data Access Object (DAO)



Person_TDG




```php
require_once('../03-singleton/DB2Connection.php');

class Person_TDG {

    public function __construct( )  {  } // empty constructor

    public function insert( $data ) {
        $conn = DB2Connection::getInstance();

        $insert_sql =
        "insert into person (firstName, lastName, dateOfBirth) values (?, ?, ?)";
        $bindVals = array($data['firstName'], $data['lastName'], $data['dateOfBirth'] );

        $insert_stmt = db2_prepare($conn, $insert_sql);
        if (!$insert_stmt)
            throw new Exception("Prepare INSERT failed! ". db2_stmt_errormsg());

        if (!db2_execute($insert_stmt, $bindVals))
            throw new Exception("db2_execute() failed! : ". db2_stmt_errormsg());
        $id = db2_last_insert_id($conn);
        return $id;
    }

    // more methods . . .
```


Application using Person_TDG

```php
require_once 'Person_TDG.php';
$bob = array('firstName' => 'Bob', 'lastName' => 'Smith',
    'dateOfBirth' => '1975-08-13');

$personGateway = new Person_TDG();
$personGateway->delete_all();   // Start with empty table

// Insert rows in person table
$bobID = $personGateway->insert($bob);
$bob['id'] = $bobID;  // save bob's id

// Update rows in person table
$bob['firstName'] = 'Robert';
$personGateway->update($bob);

// Retrieve all rows in person table
$all_rows = $personGateway->find_all();
// Retrieve specific row from person table
$row = $personGateway->find_row($bob['id']);

// Exercise domain logic
$bobsName = getFullName($bob);
$bobsBDayFmtd = formatBirthDate($bob['dateOfBirth']);
$bobsAge = calculateAge($bob['dateOfBirth']);
echo "$bobsName was born on $bobsBDayFmtd, and his age is $bobsAge";

// Delete specific row from person table
$personGateway->delete_row($tom['id']);
```

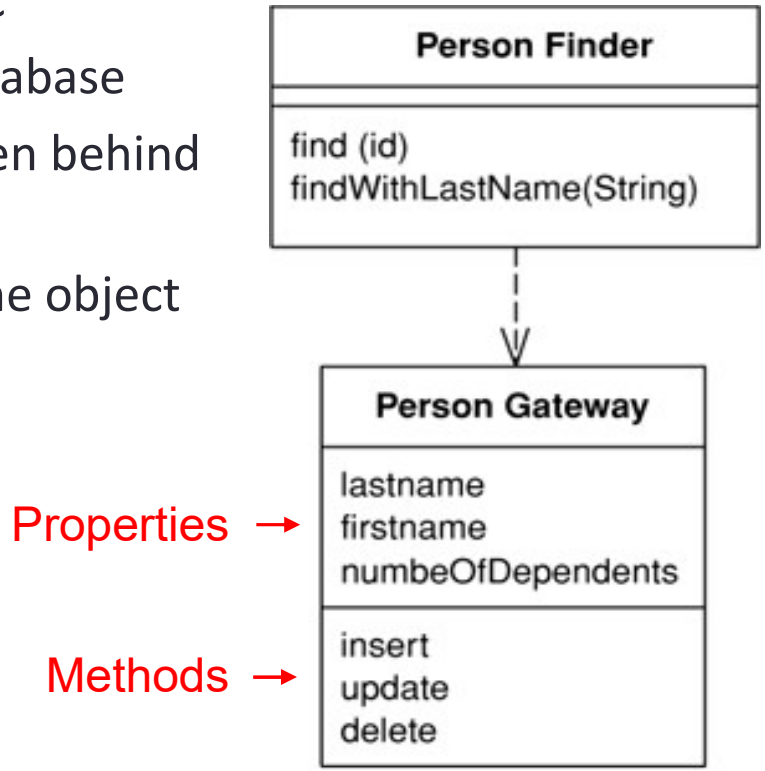

Some Common Patterns

- Object Generation Patterns
  - Singleton
  - Factory
- ORM / Database Application Patterns
  - Table Data Gateway
  - Row Data Gateway
  - Active Record
  - Data Mapper
  - Metadata Mapper
  - Query Builder

Row Data Gateway (RDG)

*An object that acts as a Gateway to a **single record** in a data source. There is **one instance per row**.*

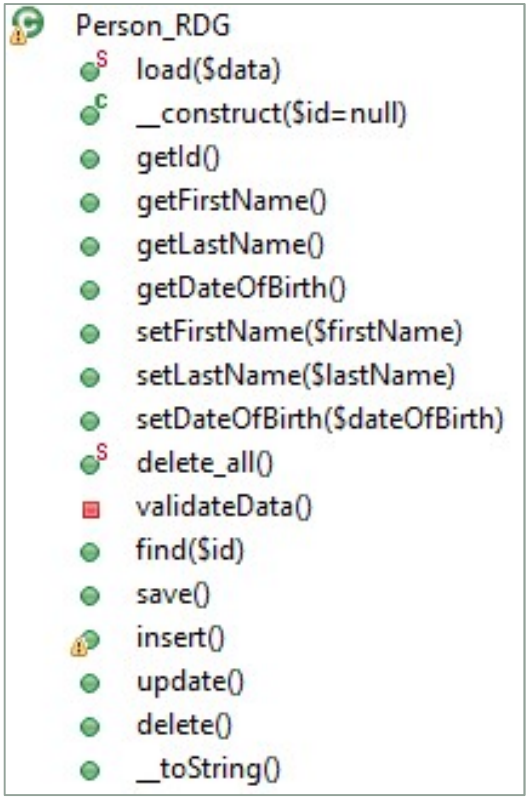
- Contains **properties** and data access logic
- Objects look exactly like the record in database
- All details of data source access are hidden behind this interface.
- Each column in the database becomes one object property.
- Where to put the find operations
  - They need to generate new object instances
    - Static find methods in gateway class?
    - Separate finder objects? = factory pattern
- RDG contains no domain logic
- RDG somewhat tedious to write,
  - good candidate for code generation using Metadata Mapping



Person_RDG

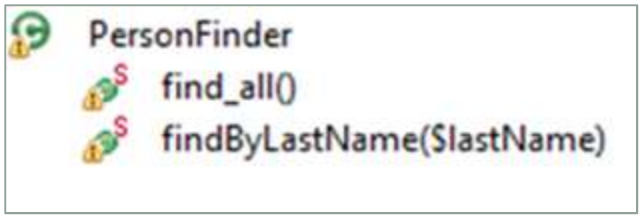
```php
require_once('../03-singleton/DB2Connection.php');

class Person_RDG {

    private $id;
    private $firstName;
    private $lastName;
    private $dateOfBirth;

    /* static Factory method: load($data)
     * Receives a database row array.
     * Returns a new Person_RDG object instance. */
    public static function load(array $data) {
        $person = new Person_RDG;
        $person->id = $data['ID'];
        $person->setFirstName($data['FIRSTNAME']);
        $person->setLastName($data['LASTNAME']);
        $person->setDateOfBirth($data['DATEOFBIRTH']);
        return $person;
    }
```

Class properties match database record fields



Factory method: generates a new object instance

PersonFinder class




```php
require_once 'Person_RDG.php';
require_once('../03-singleton/DB2Connection.php');

class PersonFinder {
    /* static Factory method - generates a list of objects */
    public static function find_all() {
        $conn = DB2Connection::getInstance();
        $sql = "select * from person";

        $stmt = db2_prepare($conn, $sql);
        db2_execute($stmt);

        // Build array of Person_RDG objects
        $obj_list = array();
        while ($row = db2_fetch_assoc($stmt )) {
            $obj_list[] = Person_RDG::load($row);
        }
        // Return collection of Person_RDG objects
        return $obj_list;
    }
```


Application using Person_RDG

```php
require_once 'Person_RDG.php';
require_once 'PersonFinder.php';

// Insert person row
$bob = new Person_RDG();
$bob->setFirstName('Bob');
$bob->setLastName('Smith');
$bob->setDateOfBirth('1975-08-13');
$bob->save();

// Update row in person table
$bob->setFirstName('Robert');
$bob->update();

// Select all rows
$all_persons = PersonFinder::find_all();

// Delete specific row from person table
$bob->delete();
```


Some Common Patterns

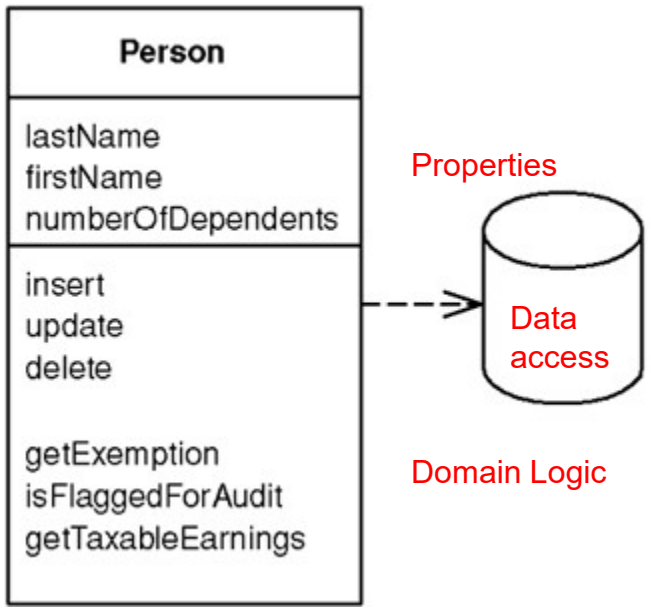
- Object Generation Patterns
  - Singleton
  - Factory

- ORM / Database Application Patterns
  - Table Data Gateway
  - Row Data Gateway
  - Active Record
  - Data Mapper
  - Metadata Mapper
  - Query Builder

Active Record pattern

*An object that wraps a row in a database table or view, encapsulates the database access, and **adds domain logic** on that data.*

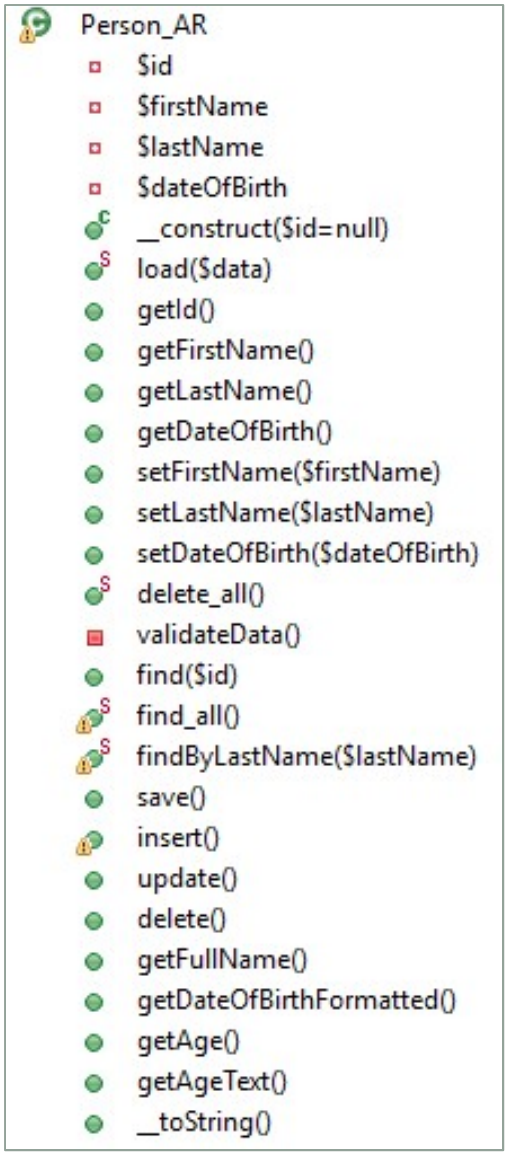
- Contains properties, data access methods, and **domain logic methods**
  - Very similar to Row Data Gateway, but adds domain logic
  - Objects look exactly like the record in database
- OK if domain logic is fairly simple, based around a single record
- Ties object structure to database design.
- As application complexity increases, better to let object design and database design evolve separately
  - => Data Mapper pattern (next)



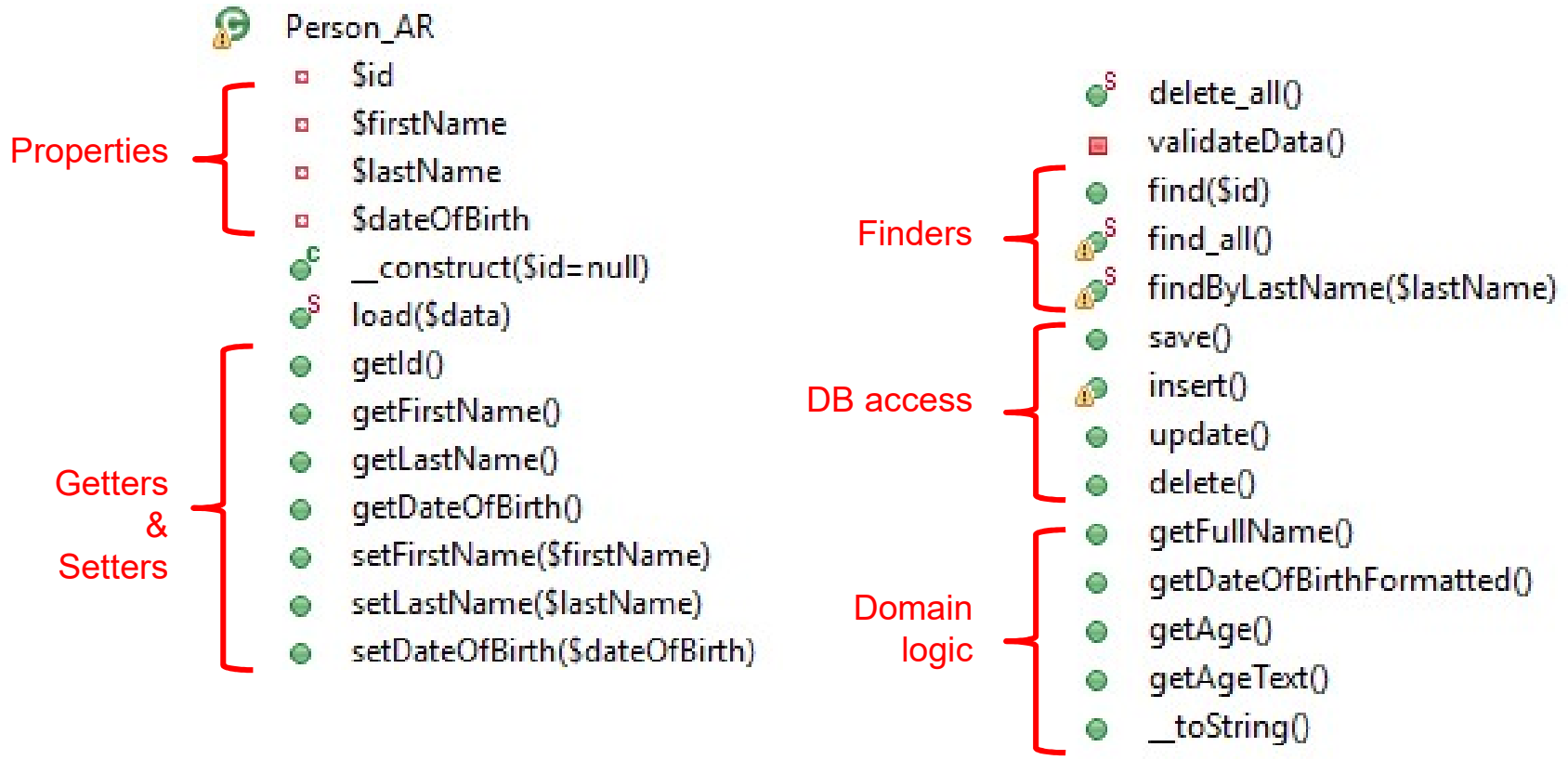
Person_AR

The *Active Record* class typically has methods that do the following:

- Construct an instance of the *Active Record* from a SQL result set row
- Construct a new instance for later insertion into the table
- Static finder methods to wrap commonly used SQL queries and return *Active Record* objects
- Update the database and insert into it the data in the *Active Record*
- Get and set the fields (convert from SQL-oriented types to better in-memory types)
- Implement some pieces of business logic



Person_AR



Application using Person_AR

```php
require_once 'Person_AR.php';

Person_AR::delete_all(); // Start with empty table

$bob = new Person_AR();
$bob->setFirstName('Bob');
$bob->setLastName('Smith');
$bob->setDateOfBirth('1975-08-13');
$bob->save();

// Update rows in person tab
$bob->setFirstName('Robert')
$bob->save();

// Select all rows
$all_persons = Person_AR::find_
foreach ($all_persons as $perso
    echo "name = {$person->getF llName()}<br>";
}

// Display age information
echo '<br>'.$bob->getAgeText();

// Delete specific row from person table
$bob->delete();
```

Active record very much like Row Data Gateway

Main difference = added domain logic in AR class

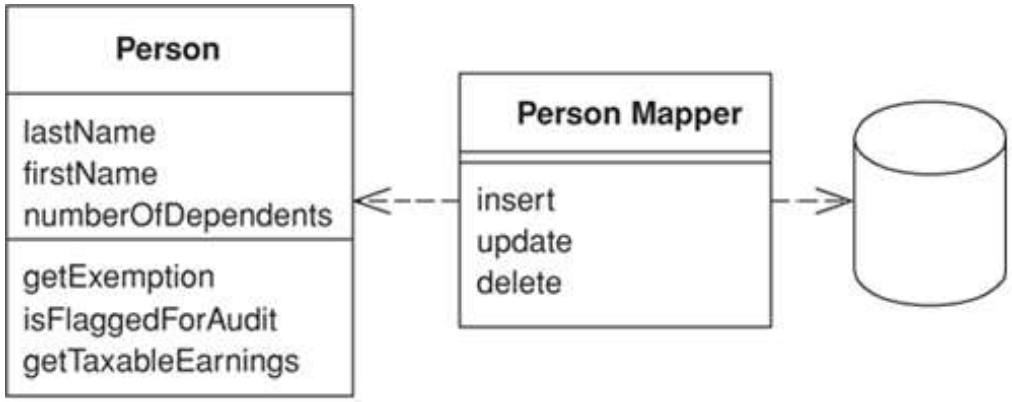
Some Common Patterns

- Object Generation Patterns
  - Singleton
  - Factory

- ORM / Database Application Patterns
  - Table Data Gateway
  - Row Data Gateway
  - Active Record
  - Data Mapper
  - Metadata Mapper
  - Query Builder

Data Mapper pattern

*A **Layer of Mappers that moves data between objects and a database** while keeping them independent of each other and the mapper itself.*



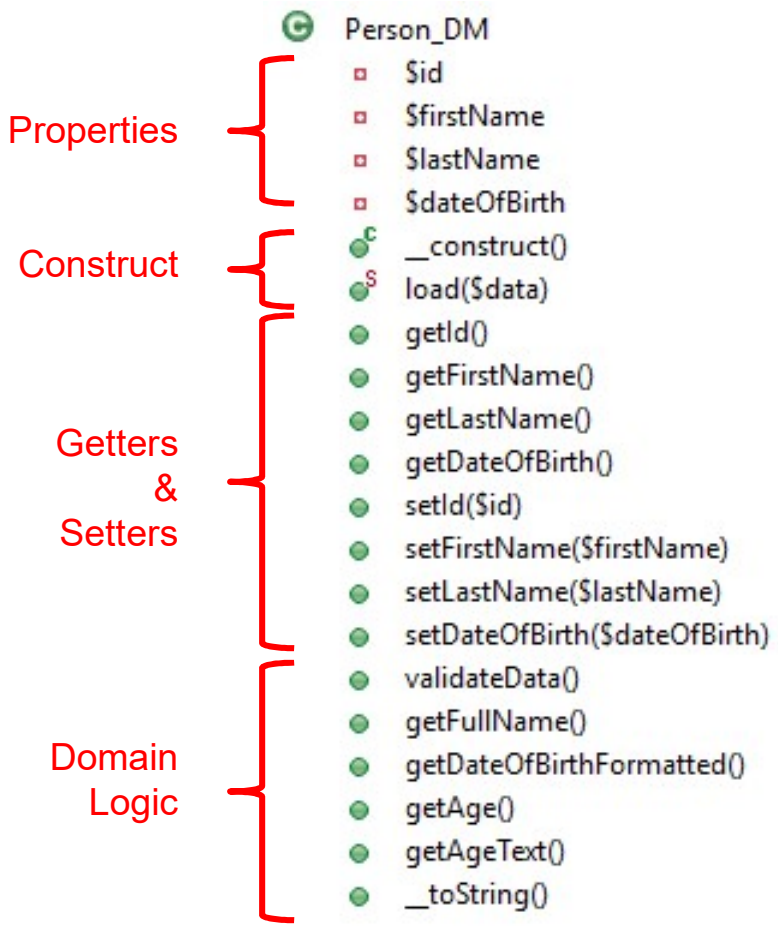
- Adds a new layer of mapping objects that handle all interactions between the object model and the database.
- Mappings can be simple or complex

- Adds one extra object per table
  - 1-to-1 cardinality of objects/tables can evolve over time
- More difficult to understand system

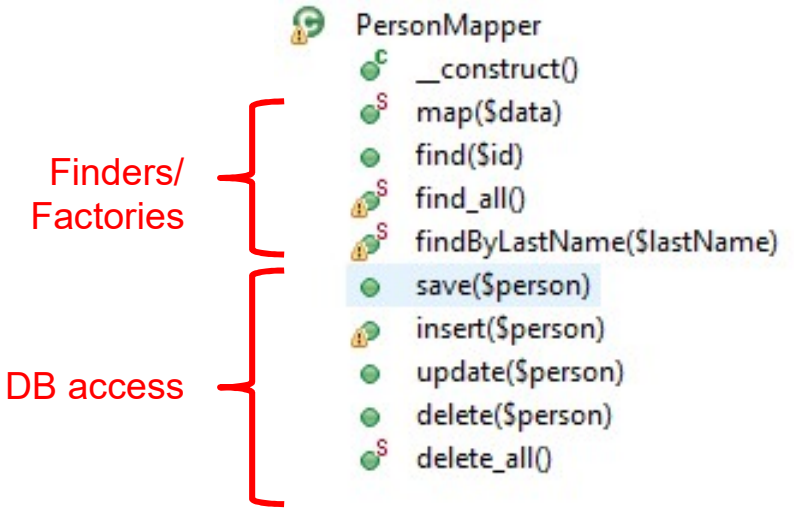
Main benefit:

- **Allows database schema and object model to evolve independently**
  - Increases abstraction
  - Leverage capabilities of OO

Person_DM



Person_Mapper



Application using Person_DM

```php
require_once 'Person_DM.php';
require_once 'PersonMapper.php';

$mapper = new PersonMapper;

$bob = new Person_DM();
$bob->setFirstName('Bob');
$bob->setLastName('Smith');
$bob->setDateOfBirth('1975-08-13');
$mapper->save( $bob );

// Update rows in person table
$bob->setFirstName('Robert');
$mapper->save( $bob );

// Select all rows
$all_persons = $mapper->find_all();
foreach ($all_persons as $person) {
    echo "name = {$person->getFullName()}<br>";
}

// Display age information
echo $bob->getAgeText();

$mapper->delete($tom);
```


Some Common Patterns

- Object Generation Patterns
  - Singleton
  - Factory

- ORM / Database Application Patterns
  - Table Data Gateway
  - Row Data Gateway
  - Active Record
  - Data Mapper
  - Metadata Mapper
  - Query Builder

Metadata Mapper pattern

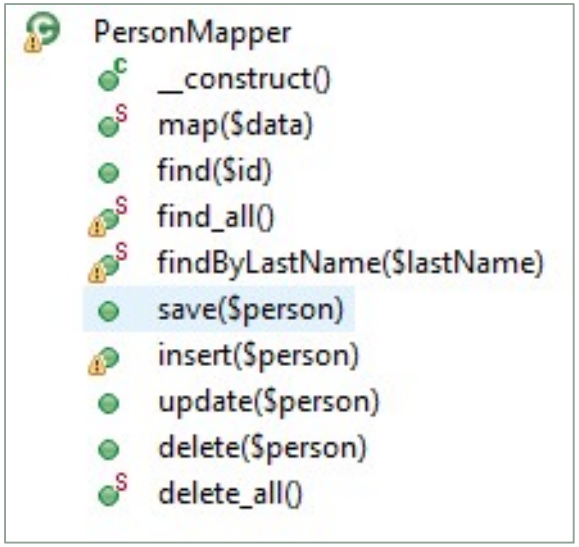
Writing mappers for every table is repetitive and tedious.
*We can **automate mapping using a table of metadata**.*

- Most code in ORM deals with
  - mapping database field names onto object property names, and
  - performing database CRUD.
- This can be tedious, and worse, ***repetitive*** (which can complicate maintenance)
- If only we could refactor this pattern into a class by itself…

Let's take another look at that PersonMapper class!

Can We Create a Generic Mapper?

What are the variables?



- `map()`: needs CLASS name (to generate objects)
- `find*()`
  select * from TABLE where KEYFIELD = KEYVALUE
- `insert()`
  insert into TABLE (LIST OF FIELDS) values (FIELD VALUES)
- `update()`
  update TABLE Set FIELD1 = VALUE1, FLD2 = VAL2
  where KEYFIELD = KEYVALUE
- `delete()`
  delete from TABLE where KEYFIELD = KEYVALUE

Variables Needed for Automated Mapper:

- Class name
- Table name (& Library)
- List of Fields (& Field Values)
- List of Key Fields (& Key Field Values)

DBMapper class (generic)

Properties and Constructor

Table, Class, and KeyFields are required

```php
require_once('../DB2Connection.php');
require_once('ToolkitService.php');
require_once 'Query_Builder.php';

class DBMapper {
    private $tableName;
    private $className;
    private $keyFields = array();
    private $metaData = array();
    private $hasAutoID = false;

    public function __construct(
        $className,
        $tableName,
        array $keyFields,
        $hasAutoID = false)
{
    if (empty($tableName) || empty($className) ||
        empty($keyFields) || !is_array($keyFields) ||
        count($keyFields) == 0)
    {
        throw new Exception('Class Name, Table Name, ' .
            'and at least one Key Field, are required ' .
            'to instantiate DBMapper object.');
    }

    $this->className = $className;
    $this->tableName = $tableName;
    $this->keyFields = $keyFields;
    $this->hasAutoID = $hasAutoID;

    $this->addMetaData( $tableName );
}
```

Use IBM Toolkit to add metadata

Metadata from DB2 system catalog

- qsys2.SYSCOLUMNS
  - View of all columns (fields) in all tables (PFs), system-wide
- Fields include:
  - Schema, Table, Column, Data type, Length, Description, etc.

```
18 select * from qsys2.SYSCOLUMNS
19 where table_schema = 'JVALANCE' and system_table_name = 'PERSON';
```

COLUMN_NAME	TABLE_NAME	TABLE_OWNER	ORDINAL_POS...	DATA_TYPE	LENGTH	NUMERI...	IS_NULL
ID	PERSON	JVALANCE	1	INTEGER	4	0	N
FIRSTNAME	PERSON	JVALANCE	2	CHAR	30	-	Y
LASTNAME	PERSON	JVALANCE	3	CHAR	40	-	Y
DATEOFBIRTH	PERSON	JVALANCE	4	DATE	4	-	Y

- We can use this to automate data mapper behavior
  - We could even add data type/length validations

addMetaData() method

```
private function addMetaData($table) {
    // Use toolkit to find table's library using RTVOBJD
    $conn = DB2Connection::getInstance();
    $tk = ToolkitService::getInstance($conn, DB2_I5_NAMING_ON);
    $CLCmd = "RTVOBJD OBJ(*LIBL/$table) OBJTYPE(*FILE) RTNLIB(?)";
    $objd = $tk->ClCommandWithOutput($CLCmd);
    $schema = $objd['RTNLIB'];

    // Query the SYSCOLUMNS table in system catalog
    $sql = "select * from qsys2/syscolumns
            where table_schema = '$schema'
            and system_table_name = '$table' ";
    $stmt = db2_prepare($conn, $sql);
    db2_execute($stmt);

    // Add each column's metadata to the master metadata array
    while ($sysColumn = db2_fetch_assoc($stmt)) {
        $colName = $sysColumn['COLUMN_NAME'];
        $this->metaData[$colName] = $sysColumn;
    }
}
```


update() method from PersonMapper

- This is from the previous pattern, Data Mapper
- Compare with Metadata Mapper update() method on next slide…

```
public function update( Person_DM $person ) {
    $person->validateData();
    $conn = DB2Connection::getInstance();

    $update_sql = "update person set firstName = ?, lastName = ?,
                                     dateOfBirth = ? where id = ?";
    $bindVals = array($person->getFirstName(), $person->getLastName(),
                      $person->getDateOfBirth(), $person->getId());

    $update_stmt = db2_prepare($conn, $update_sql);
    if (!$update_stmt) throw new Exception("Prepare UPDATE failed!");

    if (!db2_execute($update_stmt, $bindVals))
        throw new Exception("db2_execute() failed!");

    return true;
}
```

Update method from DBMapper *(part 1)*

```
public function update( $object )
{
    $conn = DB2Connection::getInstance(

    $update_sql = "update {$this->tableName} set ";
    $separator = '';

    foreach ($this->metaData as $upd_col => $col_meta) {
        // Only add non key fields to update list
        if (!in_array($upd_col, $this->keyFields)) {
            // See if getter method is callable first
            $getterMethod = "get{$upd_col}";
            if ( is_callable( array($object, $getterMethod) ) ) {
                $update_sql .= "$separator $upd_col = ?";
                $separator = ', '; // field name separator
                $bindVals[] = $object->$getterMethod();
            }
        }
    }
```

- Dynamic method calls!
  (assumes isomorphic naming between DB Columns / OO Properties)

Update method from DBMapper *(part 2)*

```
    $where = $this->getWhereClause($object);
    $update_sql .= $where['whereString'];
    $bindVals = array_merge($bindVals, $where['whereValues']);

    $update_stmt = db2_prepare($conn, $update_sql);
    return db2_execute($update_stmt, $bindVals);
}
```

```
private function getWhereClause( $object ) {
    $whereClause = ' where ';
    $separator = '';
    foreach ($this->keyFields as $keyField ) {
        // Add key fields to where clause
        $whereClause .= "$separator $keyField = ?";
        $separator = ' AND ';
        $getterMethod = "get{$keyField}";
        $bindVals[] = $object->$getterMethod();
    }

    return array(
        'whereString' => $whereClause,
        'whereValues' => $bindVals
    );
}
```


DBMapper: the `map()` method

```
/*
 * The mapper method
 */
public function map( $data ) {
    // NOTE: assumes constructor has no required parameters
    $object = new $this->className;

    foreach ($this->metaData as $col => $colMeta) {
        $setterMethod = "set$col";
        $object->$setterMethod($data[$col]);
    }
    return $object;
}
```

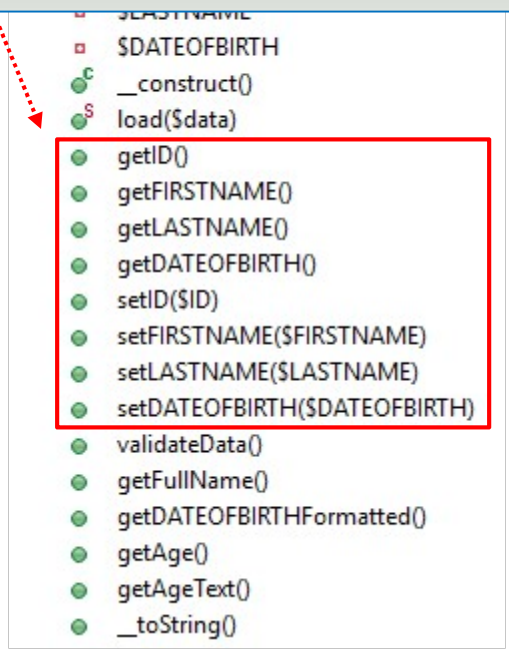
Dynamic class instantiation

Dynamic method calls

Person class (independent of mapper)

- Property names same as DB2 names
  DB2 returns field names all uppercase
- Setters/getters use uppercase property name

```
class Person {
    private $ID;
    private $FIRSTNAME;
    private $LASTNAME;
    private $DATEOFBIRTH;

    public function __construct( )  {    }

    /*
     * Static factory method -
     * create a Person object from DB data
     */
    public static function load(array $data) {
        $person = new Person;
        $person->ID = $data['ID'];
        $person->setFIRSTNAME($data['FIRSTNAME']);
        $person->setLASTNAME($data['LASTNAME']);
        $person->setDATEOFBIRTH($data['DATEOFBIRTH']);
        return $person;
    }
```



Application using DBMapper

```php
<?php
require_once 'Person.php';
require_once 'DBMapper.php';

$personMapper = new DBMapper('PERSON', 'Person', array('ID'));
$personMapper->setHasAutoID(true);

$bob = new Person();
$bob->setFirstName('Bob');
$bob->setLastName('Smith');
$bob->setDateOfBirth('1975-08-13');
$bobsID = $personMapper->insert( $bob );
//-------------------------------------------------------------
// Use find() method
$bob2 = $personMapper->find(array("ID" => $bobsID));
//-------------------------------------------------------------
// Update rows in person table
$bob->setFirstName('Robert');
$personMapper->update( $bob );
//-------------------------------------------------------------
// Display age information
echo $bob->getAgeText();

// Delete specific row from person table
$personMapper->delete($bob);
?>
```


Some Common Patterns

- Object Generation Patterns
  - Singleton
  - Factory

- ORM / Database Application Patterns
  - Table Data Gateway
  - Row Data Gateway
  - Active Record
  - Data Mapper
  - Metadata Mapper
  - Query Builder

Query_Builder pattern

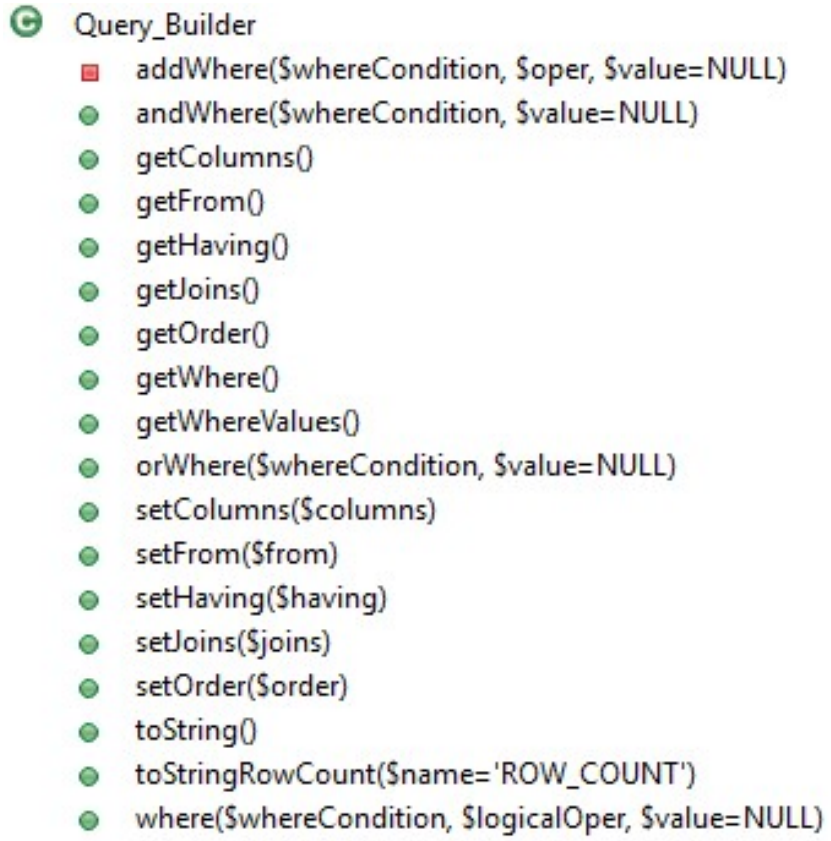
- Build SQL queries in an OO context

```
QB::select ('col1', 'col2')
 ->from ('mytable')
 ->where ('col1', 'IN', $arrayOfVals)
 ->and_where ('col2', 'IN', $arrayOfVals2)
 ->execute ();
```

Query_Builder

- Query_Builder has properties for each clause of a SQL SELECT statement

```
class Query_Builder {
    private $columns;
    private $from;
    private $joins;
    private $where;
    private $whereValues = array();
    private $order;
    private $having;
```



Query_Builder `toString()` methods

```
public function toString() {
    $cols = " * ";
    if (trim($this->columns) != '')
        $cols = $this->columns;

    $string = "Select {$cols} FROM {$this->from} ";

    if (trim($this->joins) != '')
        $string .= " $this->joins ";
    if (trim($this->where) != '')
        $string .= " WHERE $this->where ";
    if (trim($this->having) != '')
        $string .= " HAVING $this->having ";
    if (trim($this->order) != '')
        $string .= " ORDER BY $this->order ";

    return $string;
}
```

```
public function toStringRowCount (
    $name = 'ROW_COUNT' )
{
    $string = "Select COUNT(*)
                as $name FROM {$this->from} ";

    if (trim($this->joins) != '')
        $string .= " $this->joins ";
    if (trim($this->where) != '')
        $string .= " WHERE $this->where ";

    return $string;
}
```


App using Query_Builder (with DBMapper)

```php
require_once 'Customer.php';
require_once 'DBMapper.php';
require_once 'Query_Builder.php';

$customerMapper =
    new DBMapper('Customer', 'SP_CUST', array('CUST_ID'));
$customerMapper->setHasAutoID(false);

//-------------------------------------------------------
echo "<h3>Find all customers in Bahamas:</h3>";
$query = new Query_Builder();
$query->setFrom('SP_CUST');
$query->andWhere('COUNTRY = ?', 'Bahamas');
$query->setOrder('LASTNAME');

//Get row count matching query
$count = $customerMapper->find_row_count($query);
echo "<h4>$count records found for Bahamas.</h4>";

$bahamaCusts = $customerMapper->find_multi($query);
foreach ($bahamaCusts as $cust) {
    $label = $cust->getMailingLabel(true);
    echo "<p class='label'>$label</p>";
}
```


THE END

More info…

References

- Design Patterns: Elements of Reusable Object-Oriented Software
  - by: Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides (aka "*The Gang of Four*")
  - Addison-Wesley; **ISBN** 0-201-63361-2.
- Patterns of Enterprise Application Architecture
  - by Martin Fowler
  - Addison-Wesley Professional; 1 edition; **ISBN-10:** 0321127420
- Core J2EE Patterns: Best Practices and Design Strategies
  - Prentice Hall; 2nd edition; **ISBN-10:** 0321127420
- Head First Design Patterns
  - by Eric Freeman, Bert Bates, Kathy Sierra, Elisabeth Robson
  - O'Reilly Media; 1st edition; **ISBN-10:** 0596007124
- The Vietnam of Computer Science
  - Article by Ted Neward - Jun 26, 2006
  - http://blogs.tedneward.com/post/the-vietnam-of-computer-science/

Contact Information

John Valance
Division 1 Systems
47 Barrett St., So. Burlington VT 05403

johnv@div1sys.com
802-355-4024

http://www.div1sys.com

Email me for code samples!

<div1>
division 1 systems

Thank you for attending!