

Patterns in PHOOP

PHP Object-Oriented Programming Basics

Jeremy Fisher / jeremy@rentawebgeek.com / Aug 20, 2009

Agenda

- OOP Basics
- UML Class and Object Diagrams
- Design Patterns
- Antipatterns

Object-Oriented Programming and the Unified Modeling Language

Why OOP?

- Objects combine data with functionality
- Model the solution like the problem domain
- Human thought is object-oriented

Why OOP?

- Modularity
- Operation isolation
- Better code reuse
- Flexibility
- Maintainability

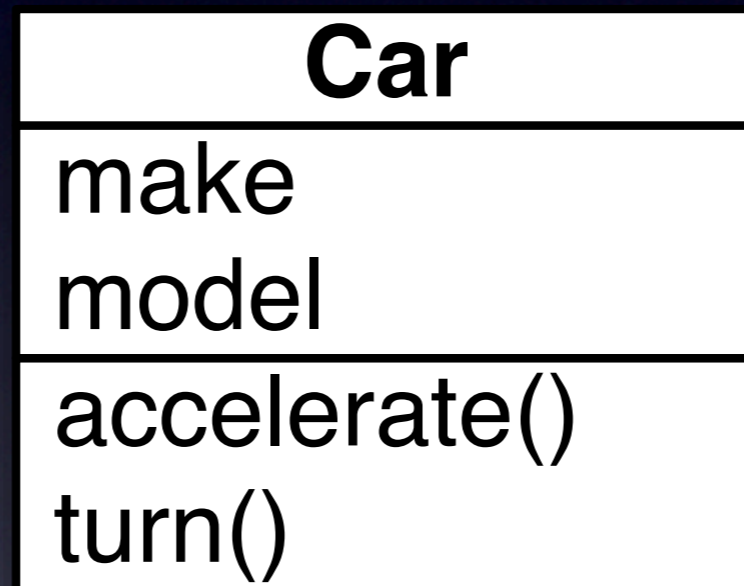
Classes

- Define the characteristics and behaviors for a thing in the system
- Nouns
- e.g. a *car* has a make and model and can accelerate and turn

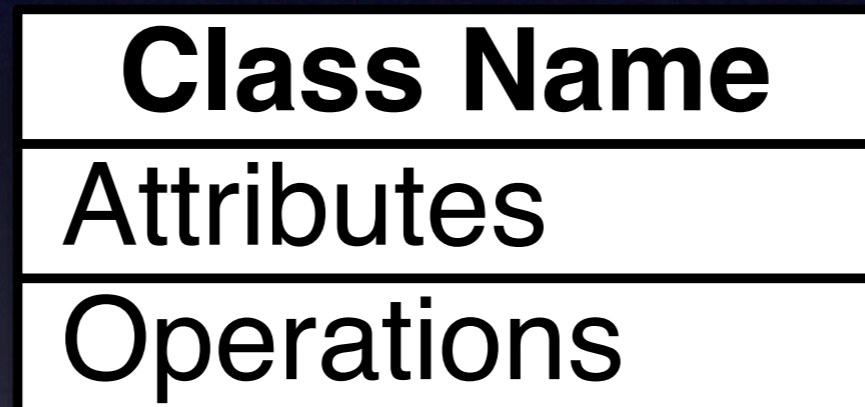
Class Example

```
class Car {  
    public $make; // Characteristic  
    public $model;  
  
    public function accelerate() { // Behavior  
        // ...  
    }  
  
    public function turn() {  
        // ...  
    }  
  
}
```

UML: Class Diagram



UML: Class Diagram



Objects

- An instance of a thing (or class).
- e.g. a particular car, such as *John's Camry*.

Object Example

```
$car = new Car();  
$car->make = 'Honda';  
$car->move();
```

UML: Object Diagram

civic : Car

: Car

Methods

- Behaviors specific to a thing
- Usually actions that objects may perform
- Also called operations
- Verbs

Members

- Characteristics of a thing
- Also called:
 - Properties
 - Variables
 - Attributes

Instance vs Static

- Instance methods always operate on the current object (PHP's default)
- Static methods are shared by all instances of a class
 - And cannot use `$this`

UML: Methods

Car

accelerate(rate : float) : void
turn(degrees : int) : void
create() : Car

Visibility

- **Public:** Visible to everything
- **Private:** Visible only within the class
- **Protected:** Visible within the class *and its subclasses*

UML: Visibility

Class Name
+publicMethod() #protectedMethod() -privateMethod()

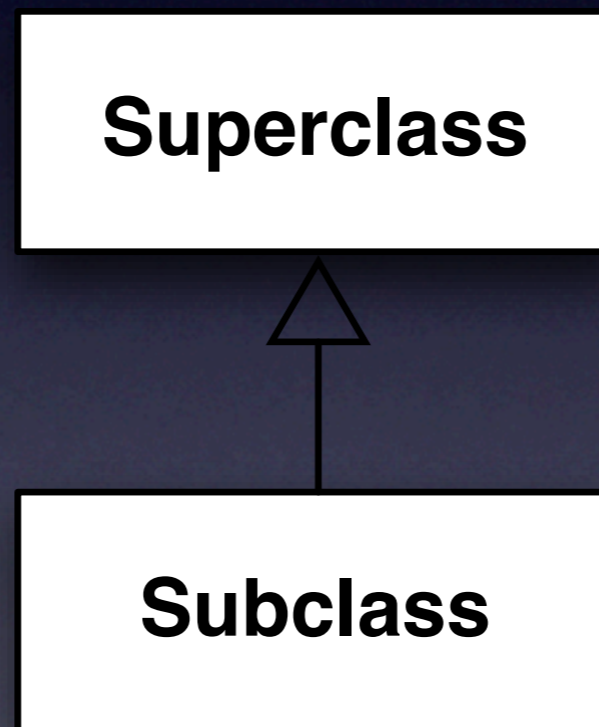
OOP Principles

- **Inheritance:** child classes inherit behaviors and characteristics from parent classes
- **Abstraction:** models may become more specific or generic as needed

OOP Principles

- **Encapsulation:** the devil may be in the details, but they're not important
- **Polymorphism:** Behaviors may change from generation to generation

UML: Inheritance



Polymorphic Methods

- **Virtual Methods:** methods that can change in child classes (PHP's default)
- **Final Methods:** methods that cannot be overridden
- **Abstract Methods:** methods that *must* be overridden

UML: Abstracts

Abstract Class

Class Name

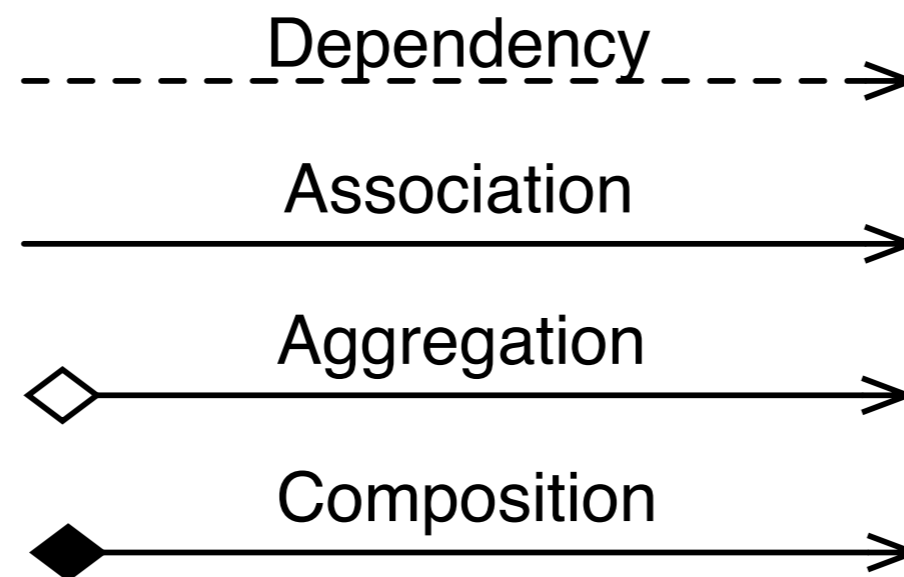
+*abstractMethod()*

+concreteMethod()

Relationships

- **Dependency:** Objects work briefly with each other
- **Association:** More prolonged interaction
- **Aggregation:** One object *references* another
- **Composition:** One class *contains* another

UML: Relationships



UML: Diagram Advice

- It's OK to mix objects and classes (and other artifacts!)
- Diagramming, not modeling
- Use diagrams for **communication**
- Don't diagram what you don't have to

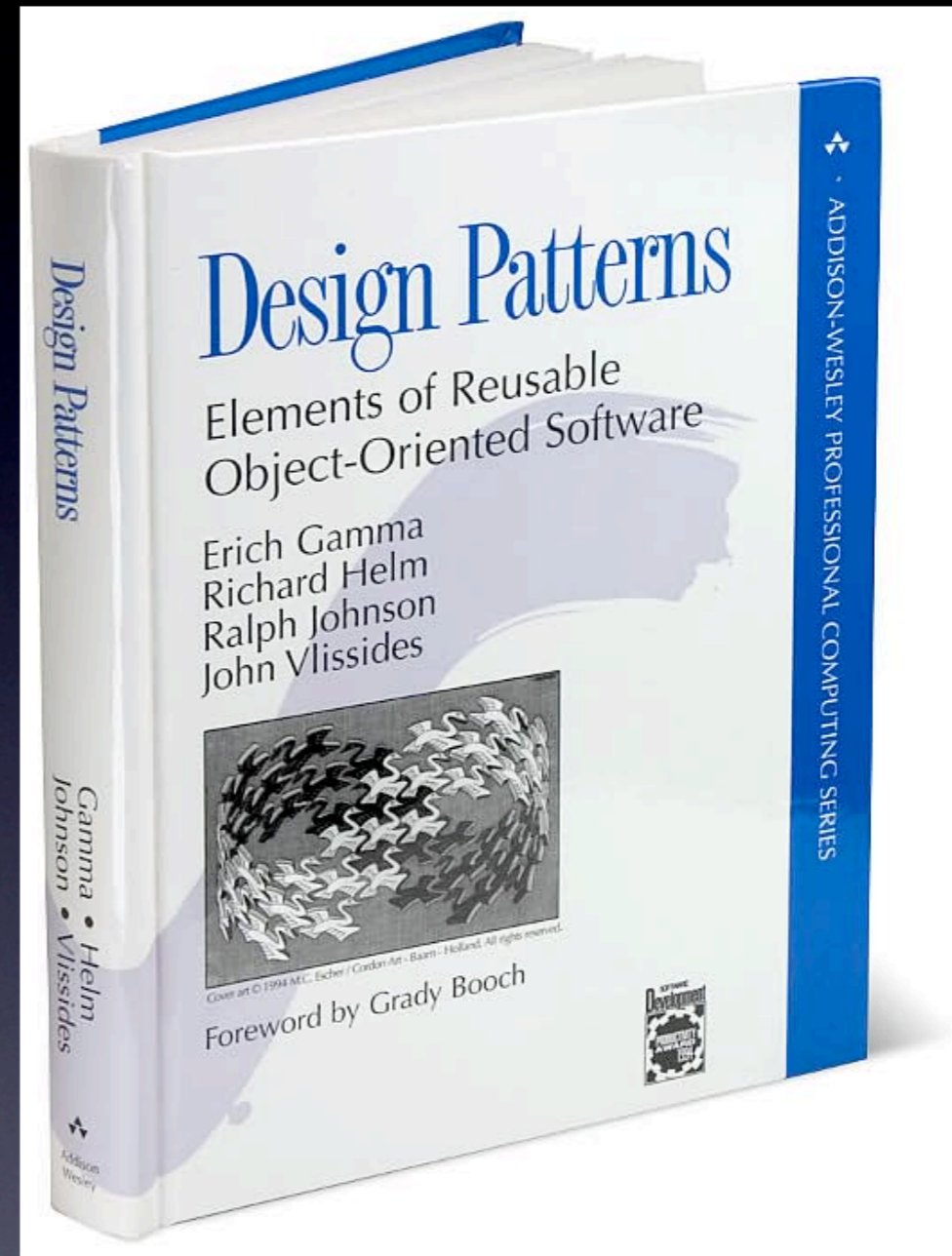
Design Patterns

Why Patterns?

- **Repeatable** solutions to common problems
- Solutions that are already **proven effective**
- **Common language** for communicating designs

Gang of Four Patterns

The original source of basic OO
design patterns



Simple Factory

- Creates objects
- May decide what type of object to create
- Useful when *new* isn't enough
- Pairs well with the strategy pattern

Car
<u>+create(model : string_) : Car</u>

Car
<u>+create() : Car</u>

Simple Factory

```
Database::create();
```

```
Database::create('reports');
```

```
Database::create()->query('select * from ...');
```

Singleton

- Allows only a **single instance** of a class
- Perhaps the best-known pattern
- Singletons are the new global

Class
- __construct()
<u>+instance() : Class</u>

There can be only one (instance)!

Command

- Encapsulate functional request as an object
- Great for common functionality across a family of operations:
 - Security
 - Logging
 - Undo history

<i>Command</i>
+success : bool
+errors : string[]
<i>+run() : bool</i>

Decorator

- Adds an effect without inheritance
- Usually inherit from the classes they decorate
- Also called a wrapper

Decorator

```
class Engine {
    public function accelerate($rate) {
        $this->speed *= $rate;
    }
}

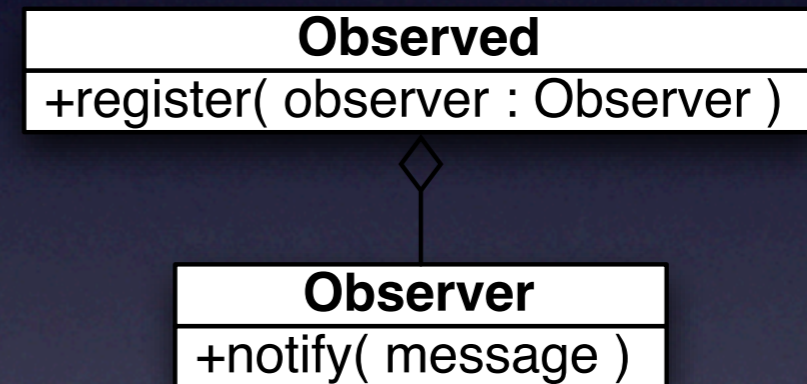
class Nitrous extends Engine {
    protected $engine = null;

    public function __construct($engine) {
        $this->engine = $engine;
    }

    public function accelerate($rate) {
        $this->engine->accelerate($rate * 1.3);
    }
}
```

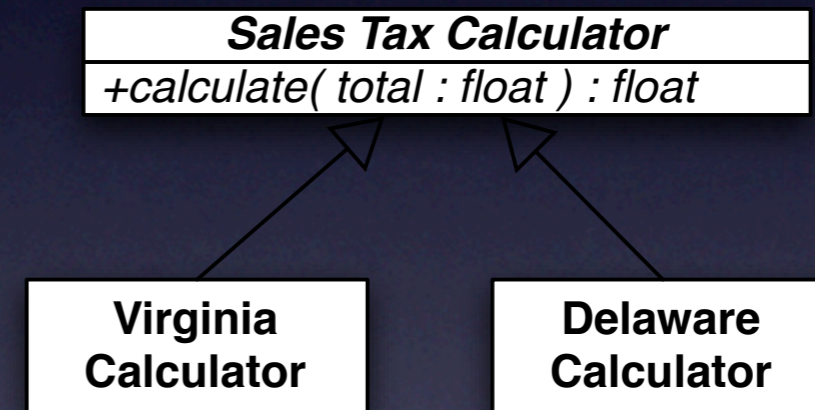
Observer

- Observes state changes
- Used for event-driven programming



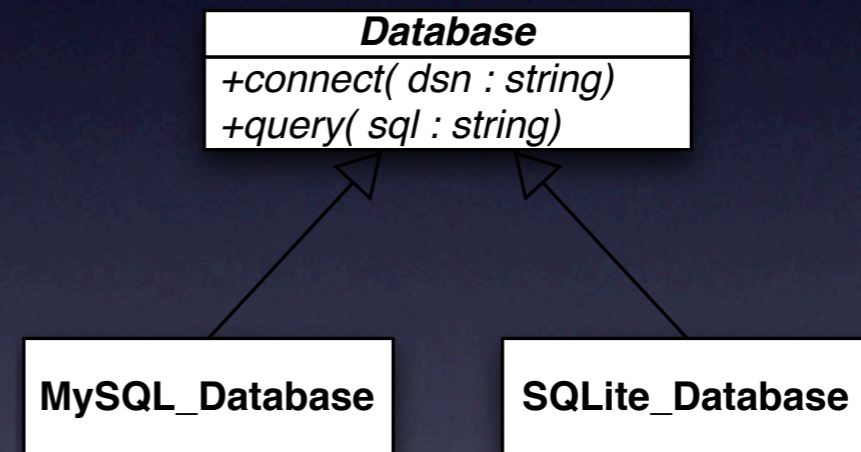
Strategy

- An interface with interchangeable implementations



Driver

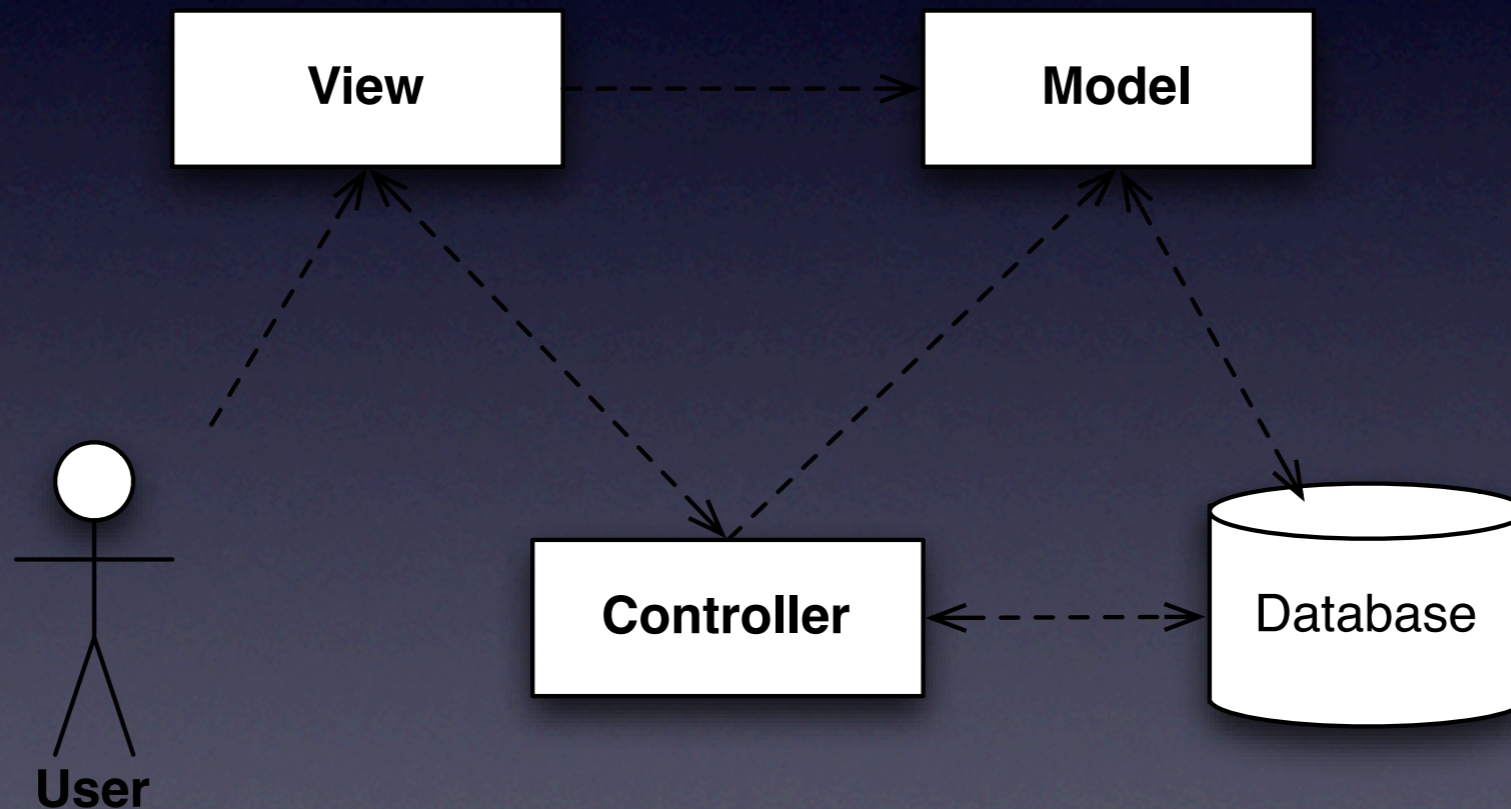
- Drivers are strategies
- Implementations needs to differ due to some *external* reason



Model View Controller

- Separates:
 - Data (model)
 - Presentation (view)
 - Business logic (controller)

Model View Controller



Model View Controller

- Examples:
 - IBeam
 - Zend
 - Cake
 - Rails
 - Struts

Controllers

- **Page Controller:** One controller per page
- **Front Controller:** One controller for everything
 - Pairs well with the Command pattern

Data Source Patterns

The Object-Relational Mapping Quagmire

Table Data Gateway

- Pro: Very simple
- Con: Tightly coupled to schema
- Con: Relationships match the physical model, not the logical

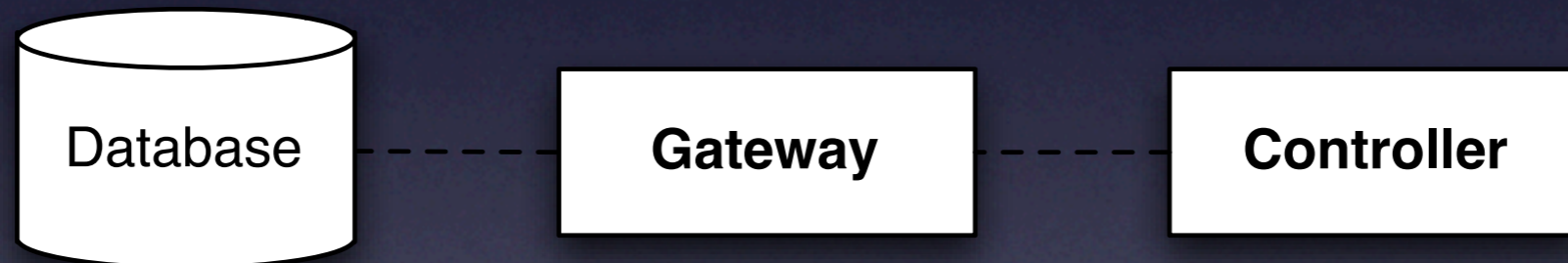
CarTableGateway
+insert(make, model, year, color)
+update(id, make, model, year, color)
+delete(id)
+find(id)
+findByMakeAndModel(make, model)

Row Data Gateway

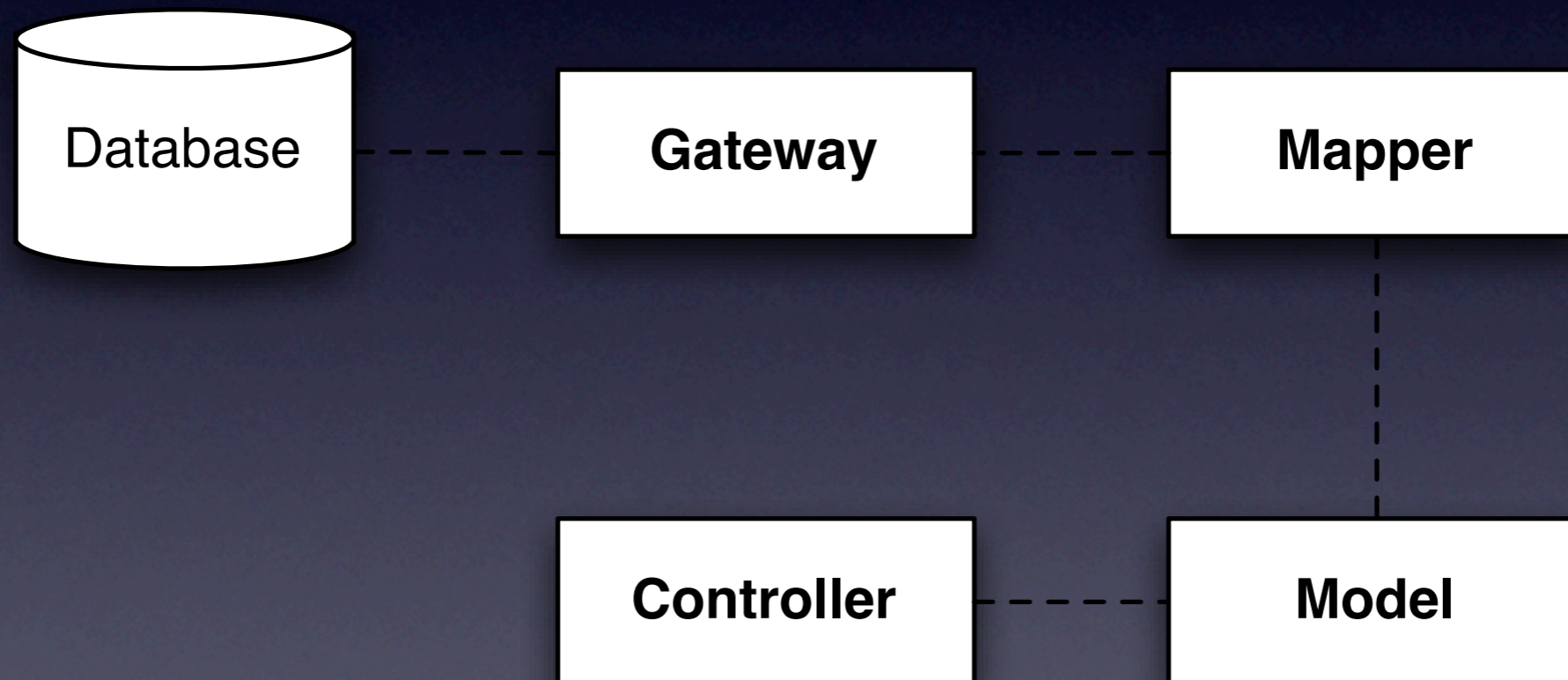
- Pro: Very simple
- Con: Tightly coupled to schema
- Con: Poor relationship support

CarRowGateway
+make
+model
+year
+id
+insert()
+update()
+delete()
<u>+find(id)</u>

Bad Gateway

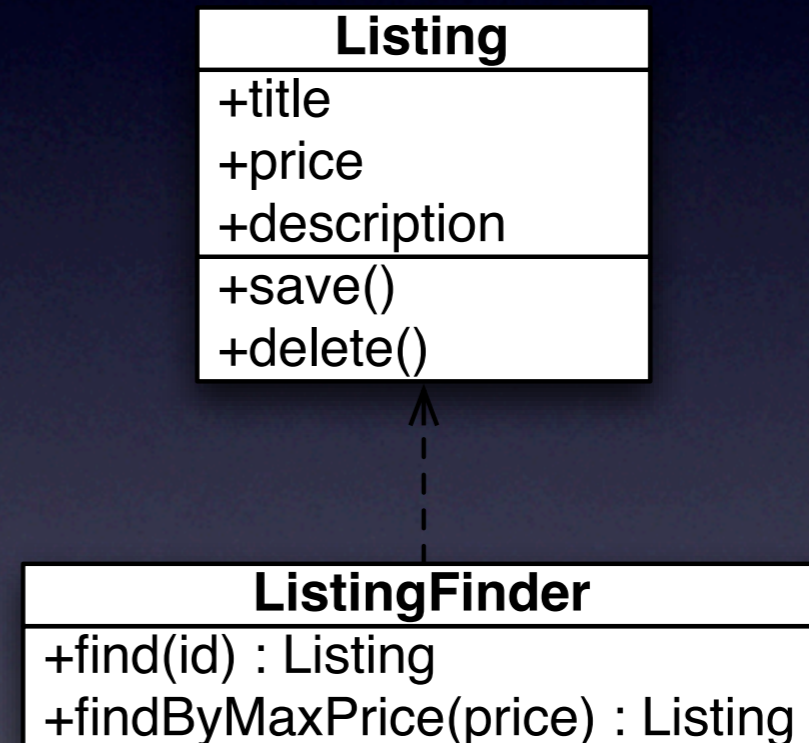


Good Gateway



Active Record

- Like RDG with business logic!
- Less tightly-coupled to the database
- Allows for more logical relationships
- Used by Rails, Doctrine, and many more

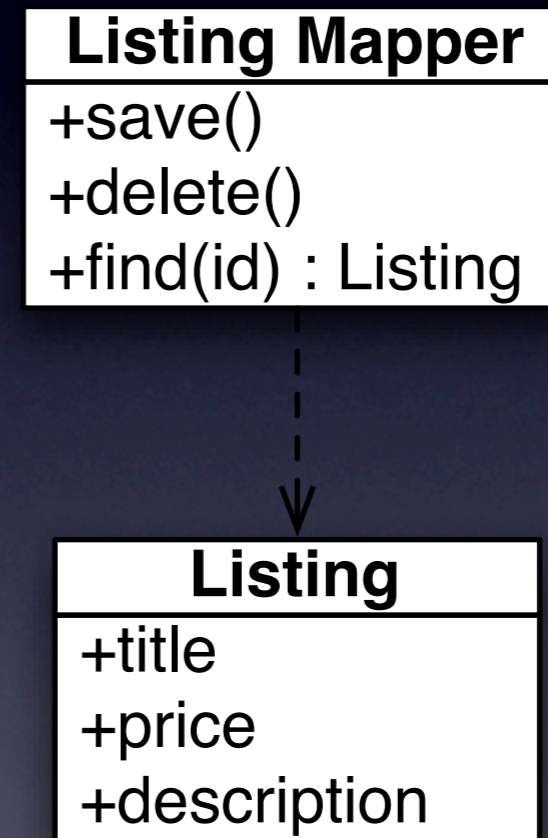


Active Record

- Con: Relationships can get tricky
- Con: Fails at complex mapping

Data Mapper

- Separates business logic from data storage
- Pairs nicely with a domain model
- Allows for complex mapping
- Very flexible, but can be complex



Antipatterns

Antipatterns

- Repeated failures rather than solutions
- Reoccurring nightmares of software engineering

“Poltergeists”

- Objects that exist only to pass information to the next

“God Object”

- One big, monolithic, untestable object that's responsible for too much.

“Blind Faith”

- Not testing a bug fix

Antipatterns

- There are many more, but try not to collect them all

Recap

- OOP is good
- Patterns are good
- Antipatterns are bad

Thanks

EOF