SOLID PHP & Code Smell Wrap-Up

LSP: Liskov substitution principle

ISP: Interface segregation principle

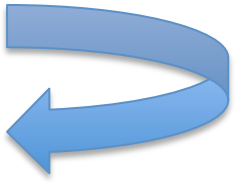
DIP: Dependency inversion principle

SOLID PHP & Code Smell Wrap-Up

LSP: Liskov substitution principle

DIP: Dependency inversion principle

ISP: Interface segregation principle



Review

- SOLID is an acronym for a set of design principles by Robert “Uncle Bob” Martin
- SRP: Single Responsibility principle – Objects should only have one responsibility
- OCP: Open/closed principle – Objects open for extension but closed for modification

LSP: Liskov substitution principle

- Objects should always be able to stand-in for their ancestors

- Named after Barbara Liskov who first introduced the principal

DIP: Dependency inversion principle

- Define interfaces from the bottom-up, not top-down

- Depend on interfaces, not on concrete classes

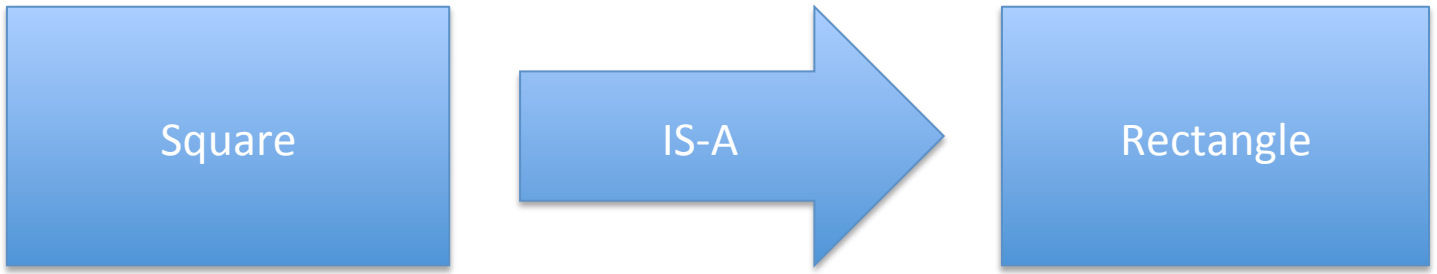
ISP: Interface segregation principle

- “many client specific interfaces are better than one general purpose interface.”

LSP Example: Rectangle & Square

- The Rectangle and Square are the “standard” example of an LSP problem.

- A Square IS-A Rectangle which happens to have the same width and height:



LSP Example: Rectangle & Square

- Lets look at an implementation and unit test

- Squares don't BEHAVE like rectangles, so they violate LSP

More Obvious LSP Violations

```
class FooMapper {
    function fetchAll() {
        return array(
        'a588ea995c5c74827a24d466d14a72101'=>'Alpha',
        'z4c853bae4a5e427a8d6b9bf33140bb2e'=>'Omega');
    }
}

class BarMapper extends FooMapper {
    function fetchAll() {
        $result = new stdClass();
        $result->a588ea995c5c74827a24d466d14a72101 = 'Alpha';
        $result->z4c853bae4a5e427a8d6b9bf33140bb2e = 'Omega';
        return $result;
    }
}
```

More Obvious LSP Violations

```
class FooView {
    function display(FooModel $foo) { /* ... */ }
}

class BarView extends FooView {
    function display(BarModel $bar) { /* ... */ }
}
```

More Obvious LSP Violations

- Removing functionality from an ancestor:

```
class DecoyDuck extends Duck
{
  function fly()
  {
    throw new Exception(
      "Decoy ducks can't fly!");
  }
}
```

Is this an LSP Violation?

```
class Foo {
    function getMapper() {
        return new FooMapper();
    }
}

class Bar extends Foo {
    function getMapper() {
        return new BarMapper();
    }
}
```

Preventing LSP Violations

- Design by Contract with Unit Tests
- Think in terms of BEHAVES-LIKE instead of IS-A when creating subclasses
- Don’t remove functionality from ancestors

Preventing LSP Violations

- Method parameters should be the same or less restrictive in what they will accept (invariant or contravariant)

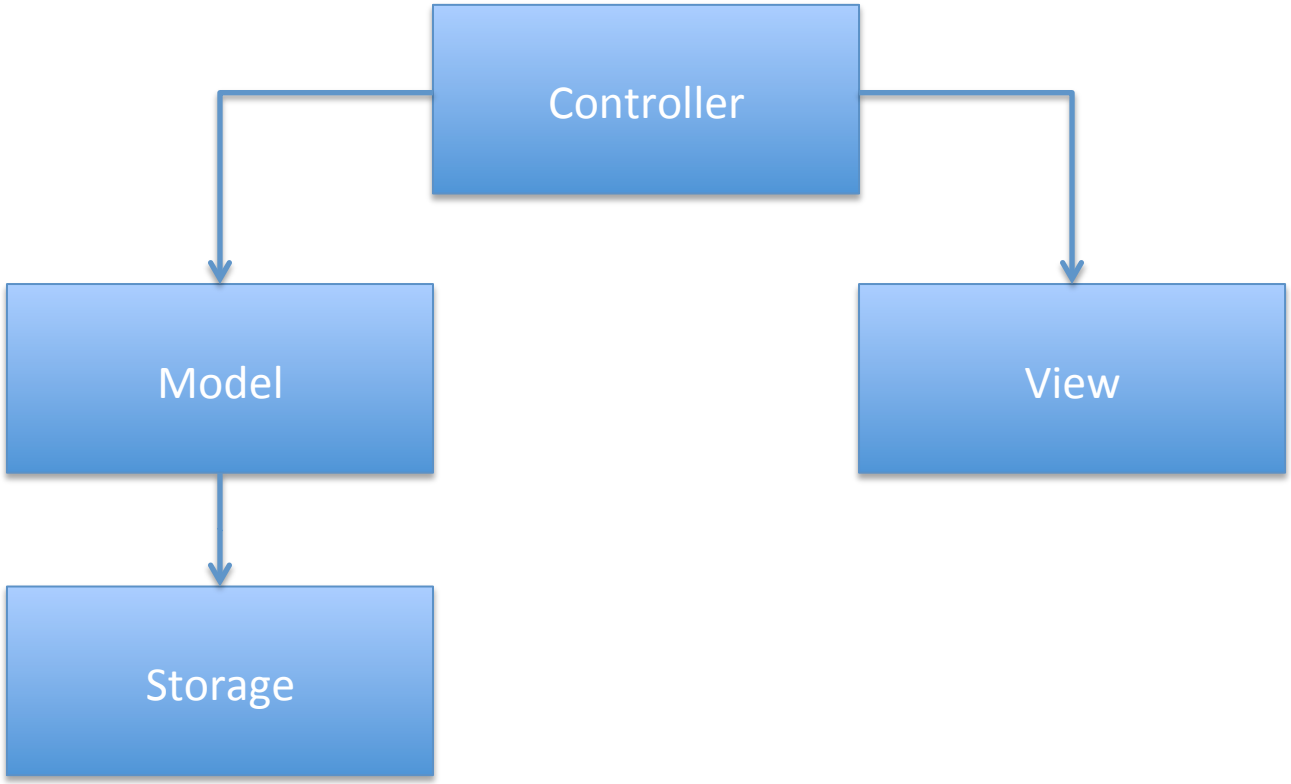
- Method return values should be the same or more restrictive in what is returned (invariant or covariant)

Smells like an LSP Violation

- Any time you need to know the type of an object (instanceof, is_a, etc)
- When the parameters or return type of an overridden method are different from its ancestor
- Throwing new exceptions

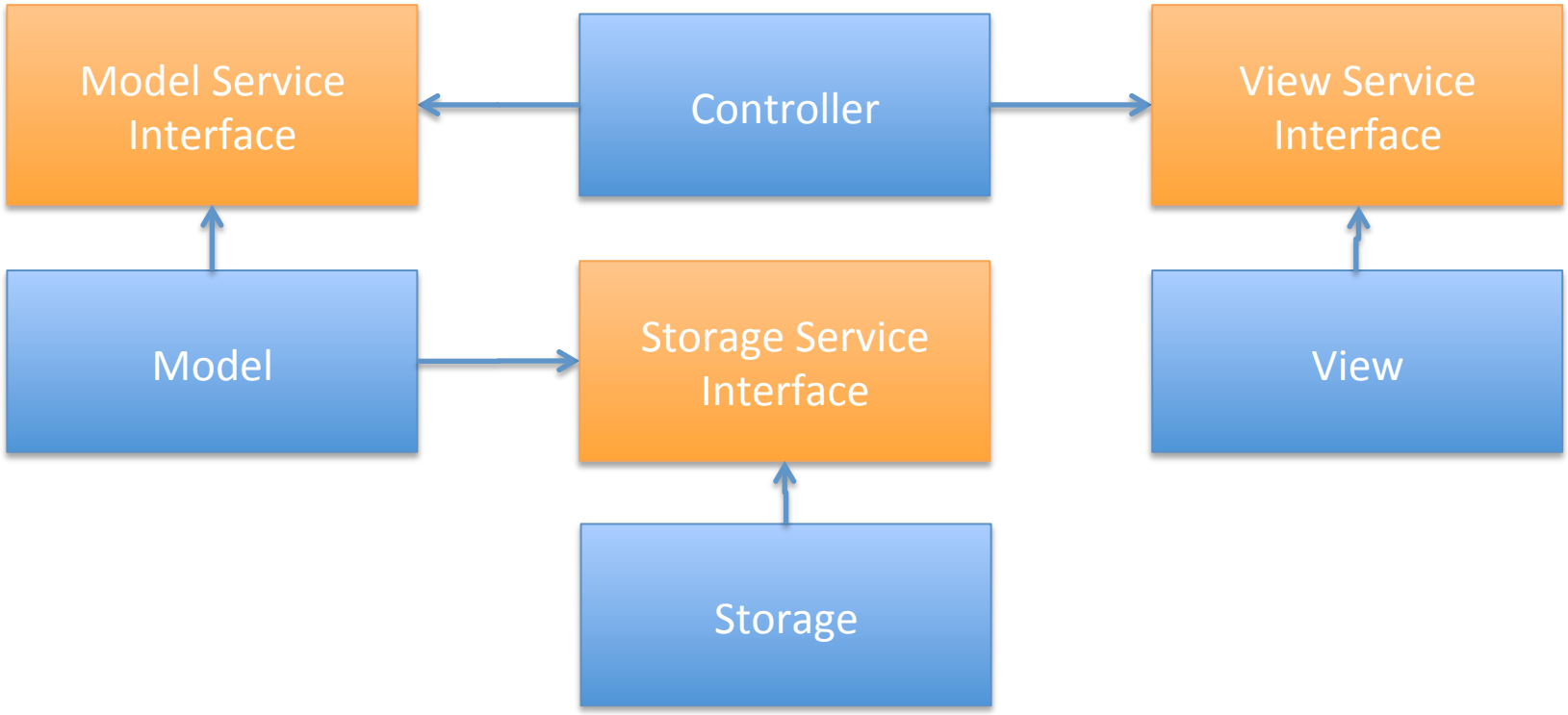
DIP: What is Inversion?

Conventional MVC Dependencies:



DIP: What is Inversion?

Inverted MVC Dependencies:



DIP Concepts

- Concrete classes only have dependencies to interfaces
- High level components build interfaces for the services they need
- Low level components depend on those high level service interfaces

Evolving towards the DIP

```
class FooControllerConventional
{
    function indexView() {
        $model = new FooModel();
        $viewData = $model->fetchAll();
        $view = new FooView();
        $view->render($viewData);
    }
}
```

Evolving towards the DIP

```
class FooControllerDI
{
	private $_model;
	private $_view;

	function __construct(FooModel $model, FooView $view) {
		$this->_model = $model;
		$this->_view = $view;
	}

	function indexView() {
		$viewData = $this->_model->fetchAll();
		$this->_view->render($viewData);
	}
}
```

Evolving towards the DIP

```
interface ModelServiceInterface {
    function fetchAll();
    //… Other required methods
}

interface ViewServiceInterface {
    function render($viewData);
    //… Other required methods
}
```

Evolving towards the DIP

```
class FooControllerDIP
{
        private $_model;
        private $_view;

        function __construct(ModelServiceInterface $model,
                                        ViewServiceInterface $view) {
                $this->_model = $model;
                $this->_view = $view;
        }

        function indexView() {
                $viewData = $this->_model->fetchAll();
                $this->_view->render($viewData);
        }
}
```

Smells like a DIP Violation

- Any dependency by one class on another concrete class: “Hard Dependency Smell”

Interface Segregation Principle

- Useful for “fat” classes. These classes:
  - May violate SRP
  - May have an “extra” method for a specific client
  - May have one responsibility which can be further segregated

ISP Example: User Model

```
class UserModel {
    /**
     * @param string $userId
     * @param string $password
     * @return string Authentication Token; empty string on failure
     */
    function doLogin($userId, $password) {}
    /**
     * @param string $token
     * @return bool
     */
    function authenticate($token) {}
}
```

ISP Example: User Model

```
interface UserLoginClient {
    function doLogin($userId, $password);
}

interface UserAuthenticationClient {
    function authenticate($token);
}
```

Smells Like an ISP Violation

- Fat Classes, “God Objects”

- Methods only used by a few clients but pollute the interface used by all clients

Thanks Folks!

- Thanks for joining me for the SOLID wrap up!
- If we have time, I have some code we can refactor to play with LSP, DIP and ISP
- Slides and code will be posted shortly
- Next week: CakePHP and Facebook Apps!