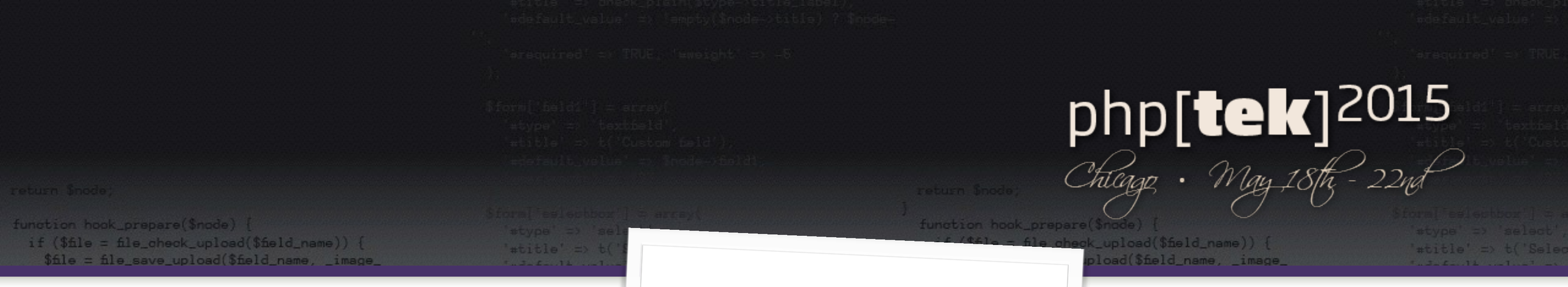
FUNDAMENTAL OBJECT ORIENTED PHP

PRESENTED BY
JEFF CAROUTH
@jcarouth



Twitter

#phptek

joind.in

joind.in/event/phptek2015

WiFi

Sheraton-MeetingRooms

Newcomers

ServerGrove Room at 5:30pm

Open Social

Atrium/Bar at 6:00pm

FUNDAMENTAL OBJECT ORIENTED PHP

PRESENTED BY
JEFF CAROUTH
@jcarouth

ABOUT THIS TUTORIAL

AGENDA

1. OOP and OOD
2. Dependencies and coupling
3. Creating interfaces
4. Sharing Behavior through Inheritance
5. Sharing Behavior through Composition
6. The SOLID Principles

WHY SHOULD WE TALK ABOUT OBJECT-ORIENTED PROGRAMMING?

We live our lives in procedural fashion. That's what makes procedural programming seem so natural.

In a procedural language, you have access to some data types, you can create variables, and you can define procedures to act upon or in response to those variables.

Procedural programming is not inherently bad. It's also not the exact opposite of OOP.

The problems with procedural code come in when code is poorly structured.

Thinking of solutions in object-oriented ways leads to improved structure.

Example: shopping cart in e-commerce application.

```php
// Procedural Shopping Cart
$product = array(
    'id' => 5634,
    'name' => 'Widget',
    'price' => 12.99
);
```

```php
// Procedural Shopping Cart
$product = array(
    'id' => 5634,
    'name' => 'Widget',
    'price' => 12.99
);

$customer = array(
    'id' => 8934512,
    'email' => 'jeff@example.com'
);
```

```php
// Procedural Shopping Cart
$product = array(
    'id' => 5634,
    'name' => 'Widget',
    'price' => 12.99
);

$customer = array(
    'id' => 8934512,
    'email' => 'jeff@example.com'
);

$cart = array();
```

```php
// Procedural Shopping Cart
$product = array(
    'id' => 5634,
    'name' => 'Widget',
    'price' => 12.99
);

$customer = array(
    'id' => 8934512,
    'email' => 'jeff@example.com'
);

$cart = array();
```

```php
function add_item_to_cart($cart, $item)
{
    if (!isset($cart['items'])) {
        $cart['items'] = array();
    }

    $cart['items'][] = $item;
    return $cart;
}
```

```php
function add_item_to_cart($cart, $item)
{
    if (!isset($cart['items'])) {
        $cart['items'] = array();
    }

    $cart['items'][] = $item;
    return $cart;
}
```

```php
function complete_purchase_of_cart($cart, $customer)
{
    $order = array(
        'line_items' => array(),
        'customer_id' => $customer['id'],
        'total' => 0.00,
    );

    foreach ($cart['items'] as $item) {
        $order['line_items'][] = $item;
        $order['total'] += $item['price'];
    }

    mail(
        $customer['email'],
        'Order of ' . count($order['line_items']) . ' items complete.'
    );
    return $order;
}
```

```php
// Procedural Shopping Cart
$product = array(
    'id' => 5634,
    'name' => 'Widget',
    'price' => 12.99
);

$customer = array(
    'id' => 8934512,
    'email' => 'jeff@example.com'
);

$cart = array();
```

```php
function add_item_to_cart($cart, $item) { /* ... */ }

function complete_purchase_of_cart($cart, $customer) { /* ... */ }
```

```php
// Procedural Shopping Cart
$product = array(
    'id' => 5634,
    'name' => 'Widget',
    'price' => 12.99
);

$customer = array(
    'id' => 8934512,
    'email' => 'jeff@example.com'
);

$cart = array();
```

```php
function add_item_to_cart($cart, $item) { /* ... */ }

function complete_purchase_of_cart($cart, $customer) { /* ... */ }

$cart = add_item_to_cart($cart, $item);
$order = complete_purchase_of_cart($cart, $customer);
```

Can we improve this with objects?

```php
// Object-Oriented Shopping Cart

class Product
{
    private $id;
    private $name;
    private $price;

    public function __construct($id, $name, $price)
    {
        $this->id = $id;
        $this->name = $name;
        $this->price = $price;
    }
}
```

```php
// Object-Oriented Shopping Cart

class Cart
{
    private $items;

    public function __construct()
    {
        $items = array();
    }

    public function addItem($item)
    {
        $this->items[] = $item;
    }

    public function sumItemPrices()
    {
        foreach ($this->items as $item) {
            $sum += $item->getPrice();
        }
    }
}
```

```php
// Object-Oriented Shopping Cart

class Customer
{
    private $id;
    private $email;

    public function __construct($id, $email)
    {
        $this->id = $id;
        $this->email = $email;
    }
}
```

```php
// Object-Oriented Shopping Cart

class Order
{
    private $items;
    private $customer;

    public function __construct($cart, $customer)
    {
        $this->items = $cart;
        $this->customer = $customer;
    }

    public function getTotal()
    {
        return $this->items->sumItemPrices();
    }
}
```

```php
// Object-Oriented Shopping Cart

class OrderProcessor
{
    public function completeOrder($order)
    {
        mail(
            $customer['email'],
            'Order of ' . $order->getNumberItems() . ' items complete.'
        );
    }
}
```

```php
// Object-Oriented Shopping Cart

class Order
{
    private $items;
    private $customer;

    public function __construct($cart, $customer)
    {
        $this->items = $cart;
        $this->customer = $customer;
    }

    public function getTotal()
    {
        return $this->items->sumItemPrices();
    }
}
```

```php
// Object-Oriented Shopping Cart

class Cart implements Countable
{
    private $items;

    public function __construct()
    {
        $items = array();
    }

    public function addItem($item) { /* ... */ }

    public function sumItemPrices() { /* ... */ }

    public function count()
    {
        return count($this->items);
    }
}
```

```php
// Object-Oriented Shopping Cart

class Order
{
    private $items;
    private $customer;

    public function __construct($cart, $customer)
    {
        $this->items = $cart;
        $this->customer = $customer;
    }

    public function getTotal() { /* ... */ }

    public function getNumberItems()
    {
        return count($this->items);
    }
}
```

```php
// Object-Oriented Shopping Cart

class OrderProcessor
{
    public function completeOrder($order)
    {
        mail(
            $customer['email'],
            'Order of ' . $order->getNumberItems() . ' items complete.'
        );
    }
}
```

```php
// Object-Oriented Shopping Cart

$product = new Product(5634, 'Widget', 12.99)
$customer = new Customer(8934512, 'jeff@example.com');
```

```php
// Object-Oriented Shopping Cart

$product = new Product(5634, 'Widget', 12.99)
$customer = new Customer(8934512, 'jeff@example.com');

$cart = new Cart();
$cart->addItem($product);
```

```php
// Object-Oriented Shopping Cart

$product = new Product(5634, 'Widget', 12.99)
$customer = new Customer(8934512, 'jeff@example.com');

$cart = new Cart();
$cart->addItem($product);

$order = new Order($cart, $customer);
```

```php
// Object-Oriented Shopping Cart

$product = new Product(5634, 'Widget', 12.99)
$customer = new Customer(8934512, 'jeff@example.com');

$cart = new Cart();
$cart->addItem($product);

$order = new Order($cart, $customer);

$orderProcessor = new OrderProcessor();
$orderProcessor->completeOrder($order);
```

What does this buy us?
Encapsulation of data.

Encapsulation can be summarized as information hiding.

Example: a bank account.

```php
// Representing a bank account

class Account
{
    private $balance;

    public function __construct($startingBalance = 0.00)
    {
        $this->balance = $startingBalance;
    }

    public function getBalance()
    {
        return $this->balance;
    }

    public function setBalance($newBalance)
    {
        $this->balance = $newBalance;
    }
}
```

```php
$account = new Account();
$currentBalance = $account->getBalance();
$account->setBalance($currentBalance + 3507.45);
var_dump($account);
```

```php
$account = new Account();
$currentBalance = $account->getBalance();
$account->setBalance($currentBalance + 3507.45);
var_dump($account);

/** output **/
object(Account)#1 (1) {
  ["balance":"Account":private]=>
  float(3507.45)
}
```

```php
function deposit($account, $amount)
{
    $newBalance = $account->getBalance() + $amount;
    $account->setBalance($newBalance);
}

$account = new Account(3507.45);
deposit($account, 492.55);
var_dump($account);
```

```php
function deposit($account, $amount)
{
    $newBalance = $account->getBalance() + $amount;
    $account->setBalance($newBalance);
}

$account = new Account(3507.45);
deposit($account, 492.55);
var_dump($account);

/** output **/
object(Account)#1 (1) {
  ["balance":"Account":private]=>
  float(4000)
}
```

```php
function withdraw($account, $amount)
{
    $newBalance = $account->getBalance() - $amount;
    $account->setBalance($newBalance);
}

$account = new Account(4000.00);
withdraw($account, 4001.99);
var_dump($account);
```

```php
function withdraw($account, $amount)
{
    $newBalance = $account->getBalance() - $amount;
    $account->setBalance($newBalance);
}

$account = new Account(4000.00);
withdraw($account, 4001.99);
var_dump($account);

/** output **/
object(Account)#1 (1) {
  ["balance":"Account":private]=>
  float(-1.9899999999998)
}
```

```php
function better_withdraw($account, $amount)
{
    if ($amount > $account->getBalance()) {
        throw new Exception(
            'Cannot withdrawal '.$amount.' from account with only ‘
                .$account->getBalance()
        );
    }

    $account->setBalance($account->getBalance() - $amount);
}

$account = new Account(4000.00);
better_withdraw($account, 4001.00);
```

```php
function better_withdraw($account, $amount)
{
    if ($amount > $account->getBalance()) {
        throw new Exception(
            'Cannot withdrawal '.$amount.' from account with only ‘
                .$account->getBalance()
        );
    }

    $account->setBalance($account->getBalance() - $amount);
}

$account = new Account(4000.00);
better_withdraw($account, 4001.00);

/** output **/
Fatal error: Uncaught exception 'Exception' with message 'Cannot withdrawal 4001 from
account with only 4000'
```

```php
// Representing a bank account

class Account
{
    private $balance;

    public function __construct($startingBalance = 0.00)
    {
        $this->balance = $startingBalance;
    }

    public function getBalance()
    {
        return $this->balance;
    }

    public function setBalance($newBalance)
    {
        $this->balance = $newBalance;
    }
}
```

```php
// Representing a bank account

class Account
{
    /* …snip… */

    public function deposit($amount)
    {
        $this->balance += $amount;
    }

    public function withdraw($amount)
    {
        if ($amount > $this->balance) {
            throw new Exception(
                'Cannot withdrawal '.$amount
                .' from account with only '.$this->balance
            );
        }

        $this->balance -= $amount;
    }
}
```

```php
$account = new Account(5.65);

$account->deposit(4.35);

$account->withdraw(11.00);
$account->withdraw(5.00);
$account->withdraw(5.00);
```

Exercise: Develop a basic set of banking objects to deal with accounts and balances.

DEALING WITH DEPENDENCIES AND COUPLING

Object-oriented programming is about managing dependencies.

Dependencies are the other objects, resources, or functions any given object uses to accomplish its responsibility.

```php
class Bank
{

    public function __construct()
    {
        $this->accountRepository = new AccountRepository();
    }

    public function openAccount($startingBalance = 0.00)
    {
        $accountNumber = $this->accountRepository->generateAccountNumber();
        $account = new Account($accountNumber, $startingBalance);
        $this->accountRepository->add($account);
        return $account;
    }
}
```

```php
class AccountRepository
{
    private $accounts;

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

    public function add($account)
    {
        $this->accounts[$account->getAccountNumber()] = $account;
    }

    public function generateAccountNumber()
    {
        do {
            $accountNumber = rand(1000, 5000);
        } while(array_key_exists($accountNumber, $this->accounts));

        return $accountNumber;
    }
}
```

```php
class Account
{
    private $accountNumber;
    private $balance;

    public function __construct($accountNumber, $balance = 0.00)
    {
        $this->accountNumber = $accountNumber;
        $this->balance = $balance;
    }

    public function getAccountNumber()
    {
        return $this->accountNumber;
    }
}
```

```php
$bank = new Bank();
$account = $bank->openAccount(50.00);
var_dump($account);
```

```php
$bank = new Bank();
$account = $bank->openAccount(50.00);
var_dump($account);

/** output **/
object(Account)#3 (2) {
  ["accountNumber":"Account":private]=>
  int(4481)
  ["balance":"Account":private]=>
  float(50)
}
```

```php
$bank = new Bank();
$account = $bank->openAccount(50.00);
var_dump($account);

/** output **/
object(Account)#3 (2) {
  ["accountNumber":"Account":private]=>
  int(4481)
  ["balance":"Account":private]=>
  float(50)
}

$anotherAccount = $bank->openAccount(1435.56);
var_dump($anotherAccount);
```

```php
$bank = new Bank();
$account = $bank->openAccount(50.00);
var_dump($account);

/** output **/
object(Account)#3 (2) {
  ["accountNumber":"Account":private]=>
  int(4481)
  ["balance":"Account":private]=>
  float(50)
}

$anotherAccount = $bank->openAccount(1435.56);
var_dump($anotherAccount);

/** output **/
object(Account)#4 (2) {
  ["accountNumber":"Account":private]=>
  int(2504)
  ["balance":"Account":private]=>
  float(1435.56)
}
```

```php
class Bank
{

    public function __construct()
    {
        $this->accountRepository = new AccountRepository();
    }

    public function openAccount($startingBalance = 0.00)
    {
        $accountNumber = $this->accountRepository->generateAccountNumber();
        $account = new Account($accountNumber, $startingBalance);
        $this->accountRepository->add($account);
        return $account;
    }
}
```

Inject dependencies where they are needed.

```php
class Bank
{
    public function __construct(AccountRepository $accountRepository)
    {
        $this->accountRepository = $accountRepository;
    }

    public function openAccount($startingBalance = 0.00)
    {
        $accountNumber = $this->accountRepository->generateAccountNumber();
        $account = new Account($accountNumber, $startingBalance);
        $this->accountRepository->add($account);
        return $account;
    }
}
```

Don’t depend on concrete classes. Depend on abstractions.

```php
class AccountRepository
{
    private $accounts;

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

    public function add($account)
    {
        $this->accounts[$account->getAccountNumber()] = $account;
    }

    public function generateAccountNumber()
    {
        do {
            $accountNumber = rand(1000, 5000);
        } while(array_key_exists($accountNumber, $this->accounts));

        return $accountNumber;
    }
}
```

```php
class InMemoryAccountRepository implements AccountRepository
{
    private $accounts;

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

    public function add($account)
    {
        $this->accounts[$account->getAccountNumber()] = $account;
    }

    public function generateAccountNumber()
    {
        do {
            $accountNumber = rand(1000, 5000);
        } while(array_key_exists($accountNumber, $this->accounts));

        return $accountNumber;
    }
}
```

```php
interface AccountRepository
{
    public function add($account);
    public function generateAccountNumber();
}
```

```php
$bank = new Bank(new InMemoryAccountRepository());
$account = $bank->openAccount(50.00);
var_dump($account);
```

```
object(Account)#3 (2) {
  ["accountNumber":"Account":private]=>
  int(3305)
  ["balance":"Account":private]=>
  float(50)
}
```

```php
$anotherAccount = $bank->openAccount(1435.56);
var_dump($anotherAccount);
```

```
object(Account)#4 (2) {
  ["accountNumber":"Account":private]=>
  int(2581)
  ["balance":"Account":private]=>
  float(1435.56)
}
```

Exercise: Improve your banking objects. Include a repository for storing and retrieving existing accounts.

CREATING INTERFACES

Simply put an interface is the collection of methods an object exposes to be interacted with.

Looking at the interface of the Account.

```php
// Representing a bank account

class Account
{
    public function __construct($startingBalance = 0.00) { /* …snip… */ }

    public function getBalance() { /* …snip… */ }

    public function deposit($amount) { /* …snip… */ }

    public function withdraw($amount) { /* …snip… */ }
}
```

```php
// Account Transactions Ledger

class Account
{
    private $balance;
    private $transactions;

    public function __construct($balance = 0.00)
    {
        $this->balance = $balance;
        $this->transactions = array();
    }

    /* snip */
}
```

```php
// Account Transactions Ledger

class Account
{
    /* snip */

    public function deposit($amount)
    {
        $this->transactions[] = 'Deposited '.$amount;
        $this->balance += $amount;
    }

    public function withdraw($amount)
    {
        if ($amount > $this->balance) {
            $this->transactions[] = 'Failed to withdraw '.$amount.' when balance '.
$this->balance;
            throw new Exception('Cannot withdraw '.$amount.' from balance '.$this-
>balance);
        }
        $this->transactions[] = 'Withdrew '.$amount;
        $this->balance -= $amount;
    }
}
```

```php
// Account Transactions Ledger

class Account
{
    /* snip */
    public function getTransactions()
    {
        return $this->transactions;
    }

    public function getBalance()
    {
        return $this->balance;
    }
}
```

```php
// Account Transactions Ledger

class Account
{
    public function __construct($balance = 0.00) { /* snip */ }
    public function deposit($amount) { /* snip */ }
    public function withdraw($amount) { /* snip */ }
    public function getTransactions() { /* snip */ }
    public function getBalance() { /* snip */ }
}
```

```php
// Account Transactions Ledger

$account = new Account(10.00);
$account->withdraw(2.86);
$account->deposit(758.34);
$account->withdraw(700.00);
try {
    $account->withdraw(66.00);
} catch (Exception $e) {
    //do nothing because I'm bad and I should feel bad
}
$account->withdraw(50.00);
var_dump($account->getTransactions());
```

```php
// Account Transactions Ledger

var_dump($account->getTransactions());

/** output **/
array(5) {
  [0]=>
  string(13) "Withdrew 2.86"
  [1]=>
  string(16) "Deposited 758.34"
  [2]=>
  string(12) "Withdrew 700"
  [3]=>
  string(40) "Failed to withdraw 66 when balance 65.48"
  [4]=>
  string(11) "Withdrew 50"
}
```

An Interface is a mechanism available in PHP to indicate that an object which implements the interface abides by the contract it specifies.

The most important job of an interface in object-oriented design is to specify the role or roles an object fulfills.

```php
// Account Transactions Ledger

interface AcceptsDeposits
{
    public function deposit($amount);
}

class Account implements AcceptsDeposits
{
    public function deposit($amount)
    {
        $this->transactions[] = 'Deposited '.$amount;
        $this->balance += $amount;
    }
}
```

BREAK TIME

SHARING BEHAVIOR THROUGH INHERITANCE

An object which obtains behaviors through its parent object is said to have inherited behavior.

Looking at our Account object, deposits and withdrawals both appear to be very similar in nature.

```php
// Account with Transactions

class Account
{
    private $balance;
    private $transactions;

    public function __construct($balance = 0.00)
    {
        $this->balance = $balance;
        $this->transactions = array();
    }

    public function postTransaction(Transaction $transaction)
    {
        try {
            $newBalance = $transaction->applyTo($this->balance);
            $this->transactions[] = $transaction;
            $this->balance = $newBalance;
        } catch (Exception $e) {
            // apply fee
        }
    }
}
```

```php
abstract class Transaction
{
    private $amount;

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

    public function applyTo($balance)
    {
        $amountToApply = $this->amount;
        if ($this->isDebit()) {
            if ($amountToApply > $balance) {
                throw new Exception('Whoops!');
            }
            return $balance - $amountToApply;
        } else {
            return $balance + $amountToApply;
        }
    }

    abstract protected function isDebit();
}
```

```php
class Withdrawal extends Transaction
{
    protected function isDebit()
    {
        return true;
    }
}

class Deposit extends Transaction
{
    protected function isDebit()
    {
        return false;
    }
}
```

```php
$account = new Account(30.00);
$account->postTransaction(new Withdrawal(5.00));
$account->postTransaction(new Deposit(75.00));
$account->postTransaction(new Withdrawal(99.99));
var_dump($account);
```

```
/** output **/
object(Account)#1 (2) {
  ["balance":"Account":private]=>
  float(0.010000000000005)
  ["transactions":"Account":private]=>
  array(3) {
    [0]=>
    object(Withdrawal)#2 (1) {
      ["amount":"Transaction":private]=>
      float(5)
    }
    [1]=>
    object(Deposit)#3 (1) {
      ["amount":"Transaction":private]=>
      float(75)
    }
    [2]=>
    object(Withdrawal)#4 (1) {
      ["amount":"Transaction":private]=>
      float(99.99)
    }
  }
}
```

SHARING BEHAVIOR THROUGH COMPOSITION

When objects are combined by holding a reference to another object to gain functionality, this is composition.

```php
// Object-Oriented Shopping Cart

class Cart
{
    private $items;

    public function __construct()
    {
        $items = array();
    }

    public function addItem($item)
    {
        $this->items[] = $item;
    }

    public function sumItemPrices()
    {
        foreach ($this->items as $item) {
            $sum += $item->getPrice();
        }
    }
}
```

```php
// Object-Oriented Shopping Cart

class Order
{
    private $items;
    private $customer;

    public function __construct($cart, $customer)
    {
        $this->items = $cart;
        $this->customer = $customer;
    }

    public function getTotal()
    {
        return $this->items->sumItemPrices();
    }
}
```

THE SOLID PRINCIPLES

Single Responsibility Principle
Open-closed Principle
Liskov Substitution Principle
Interface Substitution Principle
Dependency Inversion Principle

Single Responsibility Principle

A class should be responsible for doing one thing. It should only have one reason to change.

```php
class AccessControlManager
{
    public function __construct(Customer $customer)
    {
        $this->customer = $customer;
    }

    public function login()
    {
        if ($this->customer->authenticate()
                && $this->customer->isAuthorized()) {
            return true;
        }
        return false;
    }
}
```

```php
class Customer
{
    public function getId() {}

    public function authenticate()
    {
        // check database for user
    }

    public function isAuthorized()
    {
        // check authorization against resource
    }
}
```

Customer will change if authorization changes or if authentication changes or if the customer changes.

```php
class Customer
{
    public function getId();
}

class Login
{
    public function authenticate(Customer $customer) {
        // check customer
    }
}

class Authorize
{
    public function isAuthorized(Customer $customer) {
        //validate authorization
    }
}
```

```php
class AccessControlManager
{
    public function __construct(Customer $customer,
        Login $login,
        Authorize $authorize)
    {
        //assign
    }

    public function login()
    {
        if ($this->login->authenticate($this->customer)
                && $this->authorize->isAuthorized($this->customer)) {
            return true;
        }

        return false;
    }
}
```

This sounds easy to do. But finding and separating responsibilities is one of the hardest parts of programming.

Open-closed Principle

A software entity should be open for extension but closed for modification.

```php
class AccessControlManager
{
    public function __construct(Customer $customer,
        Login $login,
        Authorize $authorize)
    {
        //assign
    }

    public function login()
    {
        if ($this->login->authenticate($this->customer)
                && $this->authorize->isAuthorized($this->customer)) {
            return true;
        }

        return false;
    }
}
```

```php
class Login
{
    public function authenticate(Customer $customer)
    {
        // check customer
    }

    protected function getRepository() {}
}
```

```php
class Login
{
    public function authenticate(Customer $customer)
    {
        // check customer
    }

    protected function getRepository() {}
}

class LoginOauth extends Login
{
    public function authenticate(Customer $customer)
    {
        $token = $this->getAccessToken();
        // oauth login
    }

    protected function getAccessToken() {}
}
```

```php
interface LoginService
{
    public function authenticate(Customer $customer);

    protected function getRepository();

    protected function getAccessToken();
}
```

```php
class LoginDatabase implements LoginService
{
    public function authenticate(Customer $customer)
    {
        // check customer
    }

    protected function getRepository() {}
}

class LoginOauth implements LoginService
{
    public function authenticate(Customer $customer)
    {
        $token = $this->getAccessToken();
        // oauth login
    }

    protected function getAccessToken() {}
}
```

```php
class AccessControlManager
{
    public function __construct(Customer $customer,
        LoginService $login,
        Authorize $authoize)
    {
        //assign
    }

    public function login()
    {
        if ($this->login->authenticate($this->customer)
                && $this->authorize->isAuthorized($this->customer)) {
            return true;
        }

        return false;
    }
}
```

The takeaway is that your code should not need to be modified to adapt it to new situations.

Liskov Substitution Principle

Objects within an application should be able to be replaced with their subtypes without affecting the correctness of the application.

```php
class PaymentManager
{
    public function __construct(PayDateCalculator $calculator) {}

    public function schedulePayment(Payment $payment)
    {
        $payment->setPayDate($this->paydateCalculator->calculate());
        //send to db
    }
}
```

```php
class PaymentManager
{
    public function __construct(PayDateCalculator $calculator) {}

    public function schedulePayment(Payment $payment)
    {
        $payment->setPayDate($this->paydateCalculator->calculate());
        //send to db
    }
}

class PayDateCalculator
{
    public function calculate()
    {
        $today = new DateTime();
        $firstDayOfNextMonth = $today->modify('first day of next month');
        return $firstDayOfNextMonth;
    }
}
```

```php
class PayDateCalculator
{
    public function calculate()
    {
        $today = new DateTime();
        $firstDayOfNextMonth = $today->modify('first day of next month');
        return $firstDayOfNextMonth;
    }
}

class LastDayPayDateCalculator extends PayDateCalculator
{
    public function calculate()
    {
        $today = new DateTime();
        $lastDayOfMonth = $today->modify('last day of this month');
        return $lastDayOfMonth->format( 'F jS, Y' );
    }
}
```

You can’t return a DateTime from one PayDateCalculator and a String from another. That is not good.

```php
abstract class PayDateCalculator
{
    public function calculate()
    {
        $today = new DateTime();
        $payDate = $this->resolvePayDate($today);
        return $payDate;
    }

    abstract protected function resolvePayDate($today);
}

class LastDayPayDateCalculator extends PayDateCalculator
{
    protected function resolvePayDate($today)
    {
        return $today->modify('last day of this month');
    }
}

class FirstDayPayDateCalculator extends PayDateCalculator
{
    protected function resolvePayDate($today)
    {
        return $today->modify('first day of next month');
    }
```

```php
abstract class PayDateCalculator
{
    public function calculate()
    {
        $today = new DateTime();
        $payDate = $this->resolvePayDate($today);
        return $payDate;
    }

    abstract protected function resolvePayDate($today);
}
```

```php
class LastDayPayDateCalculator extends PayDateCalculator
{
    protected function resolvePayDate($today)
    {
        return $today->modify('last day of this month');
    }
}

class FirstDayPayDateCalculator extends PayDateCalculator
{
    protected function resolvePayDate($today)
    {
        return $today->modify('first day of next month');
    }
}
```

```php
class PaymentManager
{
    public function __construct(PayDateCalculator $calculator) {}

    public function schedulePayment(Payment $payment)
    {
        $payment->setPayDate($this->paydateCalculator->calculate());
        //send to db
    }
}
```

Make sure any objects which claim to implement a certain interface actually implement that interface.

Interface Segregation Principle

No client should be forced to depend on methods it doesn’t use.

```php
class Cart implements Countable
{
    private $items;

    public function __construct()
    {}

    public function addItem($item)
    {}

    public function removeItem($item)
    {}

    public function emptyCart()
    {}

    public function sumItemPrices()
    {}

    public function count()
    {}
}
```

```php
interface ShoppingCart
{
    public function addItem($item);
    public function removeItem($item);
    public function emptyCart();
    public function sumItemPrices();
    public function count();
}

class Cart implements ShoppingCart
{}
```

```php
interface ShoppingCart
{
    public function addItem($item);
    public function removeItem($item);
    public function emptyCart();
    public function sumItemPrices();
    public function count();
}

interface Repository
{
    public function addItem($item);
    public function removeItem($item);
    public function emptyCart();
}
```

```php
interface ShoppingCart
{
    public function addItem($item);
    public function removeItem($item);
    public function emptyCart();
    public function sumItemPrices();
    public function count();
}

interface Summable
{
    public function sum();
}
```

```php
interface Repository
{
    public function addItem($item);
    public function removeItem($item);
    public function emptyCart();
}

interface Summable
{
    public function sum();
}

class Cart implements Countable, Summable, Repository
{}
```

Clients of your objects should not have to depend on extraneous methods. Keep your interfaces segregated.

Dependency Inversion Principle

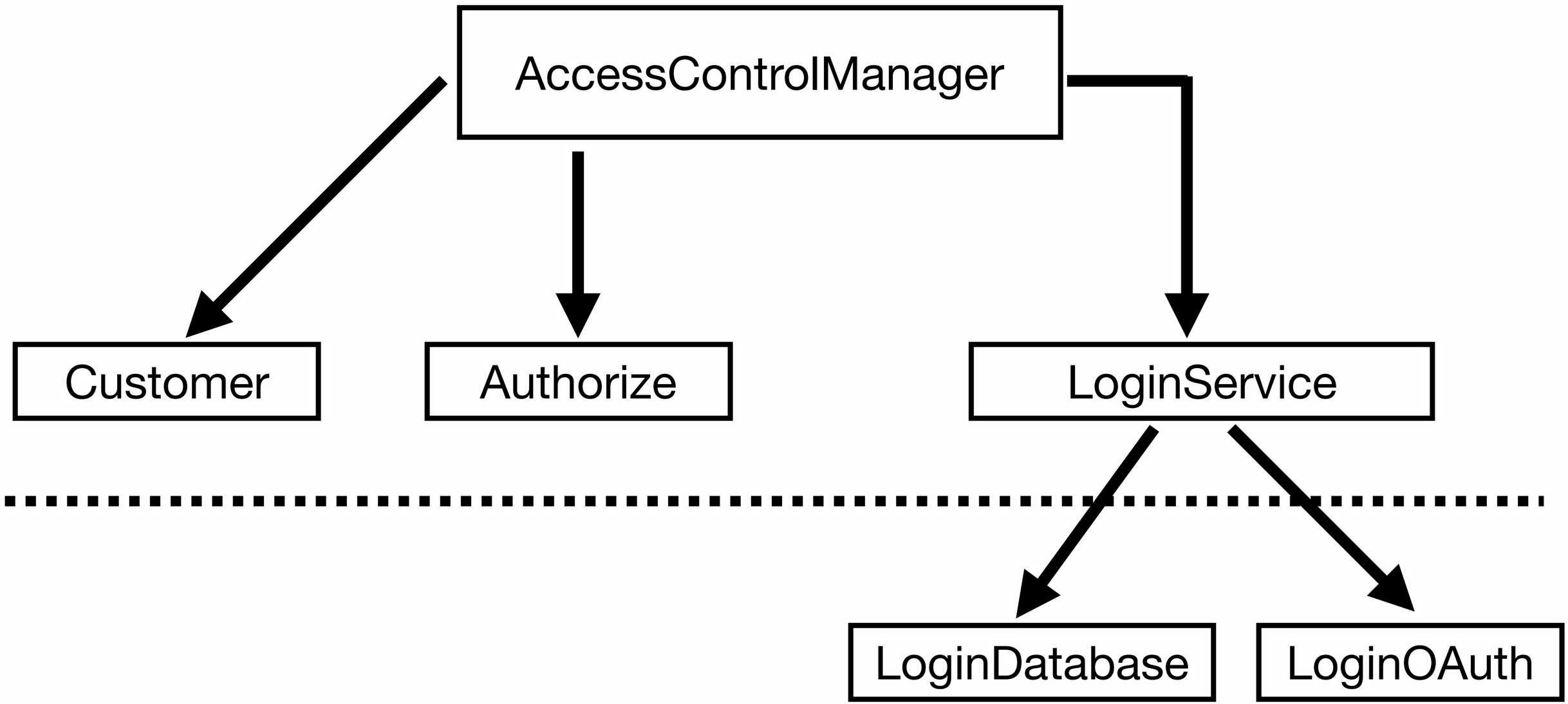
High level modules should not depend on low level modules. Both should depend on abstractions.

Abstractions should not depend on details. Details should depend on abstractions.

```php
class AccessControlManager
{
    public function __construct(Customer $customer,
        LoginService $login,
        Authorize $authoize)
    {
        //assign
    }

    public function login()
    {
        if ($this->login->authenticate($this->customer)
                && $this->authorize->isAuthorized($this->customer)) {
            return true;
        }

        return false;
    }
}
```


AccessControlManager
Customer
Authorize
LoginService
LoginDatabase
LoginOAuth

```php
class Authorize
{
    public function isAuthorized(Customer $customer)
    {
        //validate authorization
    }
}

class Customer
{
    public function getId()
    {
        return $this->id;
    }
}

interface IdentityService
{
    public function getId();
}

class Customer implements IdentityService
{
    public function getId()
    {
        return $this->id;
    }
}
```

```php
class Customer
{
    public function getId()
    {
        return $this->id;
    }
}
```

```php
class Customer
{
    public function getId()
    {
        return $this->id;
    }
}

interface IdentityService
{
    public function getId();
}
```

```php
interface IdentityService
{
    public function getId();
}

class Customer implements IdentityService
{
    public function getId()
    {
        return $this->id;
    }
}
```

```php
class AccessControlManager
{
    public function __construct(IdentityService $user,
        LoginService $login,
        Authorize $authoize)
    {
        //assign
    }

    public function login()
    {
        if ($this->login->authenticate($this->user)
                && $this->authorize->isAuthorized($this->user)) {
            return true;
        }

        return false;
    }
}
```

AccessControlManager

IdentityService

Authorize

LoginService

Customer

LoginDatabase

LoginOAuth

Depend on abstractions. Implement against abstractions. Interfaces are the key to keeping code clean.

Thank You

joind.in/13765

@jcarouth