PHP WEB PROGRAMMING: PROFESSIONAL

(Integrative-Generative AI Edition)

Design Patterns in PHP
Unit Testing and Integration Testing
Security and Performance Optimization for PHP Applications
PHP Application Deployment and Prod[illegible]ent Management
RESTful API and Microservices using PHP

References

Student Price Book Center

คำนำ

ในโลกของการพัฒนาเว็บไซต์ที่เปลี่ยนแปลงอย่างรวดเร็ว PHP ยังคงเป็นหนึ่งในภาษาที่มีบทบาทสำคัญอย่างต่อเนื่อง ทั้งในฐานะภาษาแรกที่นักพัฒนาเว็บเลือกใช้ และในฐานะเครื่องมือที่สามารถสร้างแอปพลิเคชันที่มีความซับซ้อนได้ในระดับองค์กร หนังสือ **"PHP Web Programming: Professional"** เล่มนี้ จึงถูกเขียนขึ้นเพื่อเป็นคู่มือฉบับสมบูรณ์ที่ต่อยอดจากระดับพื้นฐานไปสู่ระดับมืออาชีพ โดยเน้นแนวคิดการออกแบบซอฟต์แวร์ที่ทันสมัย การพัฒนาอย่างเป็นระบบ และการนำ PHP ไปใช้ในสถานการณ์จริงทั้งในระดับทีมและระดับองค์กร

หัวใจสำคัญของหนังสือเล่มนี้คือการพาผู้อ่านก้าวข้ามจากแนวคิดการเขียนโค้ดแบบ Procedural หรือ OOP พื้นฐาน ไปสู่การประยุกต์ใช้ **Design Patterns** เพื่อให้สามารถเขียนโค้ดที่อ่านง่าย ขยายได้ และดูแลรักษาได้ในระยะยาว โดยบทที่ 13 จะเจาะลึกแนวคิดเชิงสถาปัตยกรรม เช่น **Dependency Injection (DI)** และ **Inversion of Control (IoC)** ซึ่งเป็นรากฐานของการออกแบบระบบที่หลวม (loosely coupled) รวมถึงการจัดการกับ **Repository Pattern**, **Service Layer** และการพัฒนาแอปพลิเคชันในรูปแบบ **Event-driven** และ **Middleware** เพื่อรองรับโครงสร้างที่มีความยืดหยุ่นและขยายตัวได้ในอนาคต

จากการออกแบบระบบที่ดี ขั้นตอนต่อไปคือการรับประกันคุณภาพของระบบผ่านการทดสอบอย่างเป็นระบบ หนังสือเล่มนี้ให้ความสำคัญกับการสร้าง **Unit Testing** และ **Integration Testing** อย่างมืออาชีพ โดยใช้เครื่องมือยอดนิยมอย่าง **PHPUnit** ซึ่งถูกอธิบายไว้อย่างละเอียดในบทที่ 14 ผู้อ่านจะได้เรียนรู้เทคนิคการเขียน test ที่ครอบคลุมทั้งฟังก์ชันและคลาส การใช้ mock objects และการจัดการ test data รวมถึงการเชื่อมโยงกับแนวทาง **Continuous Integration (CI)** เพื่อให้การทดสอบกลายเป็นส่วนหนึ่งของวงจรการพัฒนาอย่างต่อเนื่อง

เมื่อโค้ดของเราพร้อม การพัฒนาแอปพลิเคชันที่มีประสิทธิภาพและปลอดภัยคือสิ่งที่หลีกเลี่ยงไม่ได้ บทที่ 15 จะพาผู้อ่านเข้าสู่โลกของ **Performance Optimization** และ **Web Security** สำหรับ PHP application ซึ่งประกอบด้วยเทคนิคการทำ Caching, การป้องกัน XSS, การเข้ารหัสและ hash password ด้วยอัลกอริธึมที่ทันสมัยอย่าง bcrypt และ Argon2 รวมถึงการป้องกันภัยคุกคามทางเว็บรูปแบบอื่น เช่น SQL Injection และ Session Hijacking พร้อมแนวทางการบังคับใช้ HTTPS อย่างถูกต้อง

ในส่วนของการ **Deploy สู่ Production** นั้น หนังสือเล่มนี้ให้รายละเอียดครบถ้วนในบทที่ 16 ตั้งแต่การเตรียม Web Server การใช้ Composer จัดการ Dependency การตั้งค่า Environment Variables ไปจนถึงเทคนิคการ Monitoring และ Debugging บน Production อย่างมืออาชีพ นอกจากนี้ยังยกตัวอย่างการปรับใช้จริงในสถานการณ์ที่พบบ่อยในโลกของ DevOps สำหรับ PHP

ท้ายที่สุด หนังสือเล่มนี้จะปิดท้ายด้วยหัวข้อที่มีความสำคัญในยุค API-first และระบบกระจายตัว ได้แก่ **การพัฒนา RESTful API และ Microservices ด้วย PHP** ซึ่งนำเสนอในบทที่ 17 โดยมีการวางโครงสร้าง REST API ตามหลักมาตรฐาน การจัดการ Routing, Request, Response การจัดรูปแบบข้อมูลด้วย JSON และ XML ตลอดจนแนวทางการออกแบบและเชื่อมต่อ Microservices อย่างปลอดภัย

และมีประสิทธิภาพ โดยมีตัวอย่างโปรแกรมที่สามารถนำไปใช้งานหรือประยุกต์ใช้ในโครงการจริงได้ทันที

หนังสือเล่มนี้เหมาะสำหรับนักพัฒนา PHP ที่มีพื้นฐานอยู่แล้วและต้องการยกระดับการเขียนโค้ด การออกแบบระบบ และการวางโครงสร้างแอปพลิเคชันให้สามารถใช้งานได้จริงในระดับองค์กร โดยเฉพาะผู้ที่ต้องรับผิดชอบโครงการขนาดกลางถึงใหญ่ที่ต้องการความมั่นคง ทดสอบได้ ปลอดภัย และสามารถดูแลรักษาได้ในระยะยาว

ในฐานะผู้เขียน เราหวังว่าเนื้อหาในเล่มนี้จะช่วยให้คุณมีมุมมองใหม่ๆ ในการพัฒนา PHP Application และเป็นอีกหนึ่งแรงผลักดันให้วงการ PHP ของไทยก้าวไปสู่ความเป็นมืออาชีพอย่างมั่นคง

ด้วยรักและปรารถนาดี

ศูนย์หนังสือราคานักเรียน

สารบัญ

หน้า

บทที่ 13
แนวคิด Design Patterns ใน PHP
(Design Patterns in PHP)

เนื้อหา

- แนวคิด Design Patterns ใน PHP
- แนวคิด Design Patterns ใน PHP (เชิงลึก)
- รายละเอียดเชิงลึก 4 Design Patterns
- Dependency Injection (DI) และ Inversion of Control (IoC) ใน PHP
- Repository Pattern และ Service Layer ใน PHP
- Event-driven Programming และ Middleware ใน PHP
- ตัวอย่างโปรแกรม PHP แบบบูรณาการ

บทที่ 13: แนวคิด Design Patterns ใน PHP

ในการพัฒนาแอปพลิเคชันที่มีความซับซ้อนสูงและต้องการโครงสร้างที่สามารถดูแลรักษาได้ในระยะยาว **Design Patterns** หรือ "รูปแบบการออกแบบซอฟต์แวร์" จึงเข้ามามีบทบาทสำคัญเป็นอย่างยิ่ง โดยเฉพาะในภาษา PHP ที่แม้จะเป็นภาษาสคริปต์ แต่ก็สามารถนำแนวคิดเชิงสถาปัตยกรรมมาประยุกต์ใช้เพื่อเสริมสร้างความยืดหยุ่น ความสามารถในการทดสอบ และลดการพึ่งพาโค้ดที่ซ้ำซ้อน (redundant code) ได้อย่างมีประสิทธิภาพ

บทที่ 13 นี้จะพาผู้อ่านเข้าสู่โลกของ **Design Patterns ใน PHP** โดยเริ่มต้นจากการเรียนรู้รูปแบบพื้นฐานที่ได้รับความนิยมสูง ได้แก่ **Singleton** สำหรับการควบคุมจำนวนอินสแตนซ์ของคลาส, **Factory** สำหรับการแยกความรับผิดชอบในการสร้างอ็อบเจ็กต์, **Strategy** ที่ช่วยให้สามารถสลับพฤติกรรมของคลาสได้ใน runtime และ **Observer** ซึ่งเหมาะสำหรับระบบที่ต้องแจ้งเตือนเหตุการณ์ระหว่างอ็อบเจ็กต์ต่าง ๆ

จากนั้นจะต่อยอดสู่แนวคิดระดับโครงสร้างแอปพลิเคชัน ด้วยการแนะนำ **Dependency Injection (DI)** และ **Inversion of Control (IoC)** ซึ่งช่วยลดการผูกติดกันของโค้ด (tight coupling) และเพิ่มความสามารถในการทดสอบ (testability) โดยมีตัวอย่างการเขียนคลาสที่รองรับ DI และ IoC อย่างเป็นรูปธรรมในบริบทของ PHP

หัวข้อถัดไปคือ **Repository Pattern และ Service Layer** ซึ่งเป็นแนวทางในการแยกการเข้าถึงข้อมูล (data access) ออกจาก business logic โดยการใช้ repository จะช่วยจัดระเบียบคำสั่งที่

ติดต่อฐานข้อมูล และ Service Layer จะช่วยจัดการกระบวนการเชิงธุรกิจอย่างชัดเจน ทำให้โค้ดสามารถขยายตัวได้ง่ายเมื่อมีความซับซ้อนเพิ่มขึ้น

นอกจากนี้ยังมีการแนะนำแนวทางการพัฒนาแอปพลิเคชันแบบ **Event-driven programming** ซึ่งช่วยให้ระบบสามารถตอบสนองต่อเหตุการณ์ต่าง ๆ ได้อย่างยืดหยุ่น โดยไม่ต้องผูกติดกับลำดับการทำงานที่ตายตัว รวมถึงการประยุกต์ใช้ **Middleware** ซึ่งเป็นแนวคิดที่มักใช้ร่วมกับ PHP Framework เพื่อควบคุมกระบวนการ request/response อย่างมีระบบ เช่น การตรวจสอบสิทธิ์ การบันทึก log หรือการจัดการข้อมูลก่อนเข้าสู่ controller

บทเรียนในบทนี้เน้นทั้งด้าน **แนวคิดเชิงสถาปัตยกรรม** และ **การนำไปใช้จริง** โดยมีตัวอย่างโค้ดที่ออกแบบมาอย่างรัดกุม เข้าใจง่าย และพร้อมให้ผู้อ่านนำไปประยุกต์ใช้ในโปรเจกต์ของตนเอง เพื่อให้สามารถออกแบบระบบที่มีความยั่งยืน รองรับการเปลี่ยนแปลง และสามารถทำงานร่วมกับทีมขนาดใหญ่ได้อย่างมืออาชีพ

ด้วยแนวทางในบทที่ 13 นี้ ผู้อ่านจะสามารถต่อยอดการเขียน PHP สู่ระดับการออกแบบเชิงระบบ (System Design) โดยใช้ Design Patterns เป็นเครื่องมือหลักในการยกระดับคุณภาพซอฟต์แวร์และวางรากฐานที่มั่นคงสำหรับการพัฒนาเว็บแอปพลิเคชันในระดับองค์กร.

แนวคิด Design Patterns ใน PHP

---

1. Singleton Pattern

- **นิยาม**: การจำกัดให้คลาสมีแค่หนึ่งอินสแตนซ์เดียวตลอดการทำงานของโปรแกรม
- **ประโยชน์**: ควบคุมการเข้าถึงทรัพยากรที่แชร์ เช่น การเชื่อมต่อฐานข้อมูล
- **ตัวอย่าง PHP แบบง่าย**:

```
class Singleton {
    private static $instance = null;
    private function __construct() {}
    public static function getInstance() {
        if (self::$instance === null) {
            self::$instance = new Singleton();
        }
        return self::$instance;
    }
}
```

---

2. Factory Pattern

- **นิยาม**: สร้างอินสแตนซ์ของคลาสโดยแยกความรับผิดชอบการสร้างวัตถุออกจากการใช้งาน

- **ประโยชน์**: เพิ่มความยืดหยุ่น ลดการผูกมัดโค้ดกับคลาสเฉพาะ
- **ตัวอย่าง**:

```
interface Product {
    public function getName();
}

class ProductA implements Product {
    public function getName() { return 'Product A'; }
}

class ProductFactory {
    public static function create($type) {
        if ($type === 'A') return new ProductA();
        // เพิ่มเติมประเภทอื่น ๆ
    }
}
```

---

3. Strategy Pattern

- **นิยาม**: แยกกลยุทธ์การทำงานเป็นอิสระจากกัน และสามารถสลับใช้ได้ระหว่าง runtime
- **ประโยชน์**: ยืดหยุ่นต่อการเปลี่ยนแปลงพฤติกรรมโดยไม่ต้องแก้โค้ดหลัก
- **ตัวอย่าง**:

```
interface PaymentStrategy {
    public function pay($amount);
}

class CreditCardPayment implements PaymentStrategy {
    public function pay($amount) { echo "Paying $amount by Credit Card"; }
}

class PaypalPayment implements PaymentStrategy {
    public function pay($amount) { echo "Paying $amount by PayPal"; }
}

class ShoppingCart {
```

```
    private $paymentMethod;
    public function setPaymentMethod(PaymentStrategy $method) {
        $this->paymentMethod = $method;
    }
    public function checkout($amount) {
        $this->paymentMethod->pay($amount);
    }
}
```

---

4. Observer Pattern

- **นิยาม**: กำหนดความสัมพันธ์แบบ one-to-many ระหว่างวัตถุ เพื่อให้วัตถุเปลี่ยนสถานะจะส่งการแจ้งเตือนไปยัง observer หลายตัว
- **ประโยชน์**: เหมาะสำหรับ event-driven programming หรือระบบแจ้งเตือน
- **ตัวอย่าง**:

```
interface Observer {
    public function update($data);
}

class Subject {
    private $observers = [];
    public function attach(Observer $obs) {
        $this->observers[] = $obs;
    }
    public function notify($data) {
        foreach ($this->observers as $obs) {
            $obs->update($data);
        }
    }
}
```

---

5. Dependency Injection (DI) และ Inversion of Control (IoC)

- **DI**: การจัดการ dependency ของคลาสโดยให้ external system ส่ง dependency มาแทนที่คลาสสร้างเอง

- **IoC**: หลักการที่ flow การควบคุมถูกกลับด้านให้ภายนอกควบคุม (เช่น framework หรือ container)
- **ประโยชน์**: ลดการผูกมัด ลดความซับซ้อน และเพิ่ม testability
- **ตัวอย่าง**:

```
class Mailer {}
class UserController {
    private $mailer;
    public function __construct(Mailer $mailer) {
        $this->mailer = $mailer;
    }
}
```

---

6. Repository Pattern และ Service Layer

- **Repository Pattern**: แยกชั้นจัดการ data access ออกจาก business logic
- **Service Layer**: ชั้นกลางที่รวบรวม business logic และประสานงาน repository
- **ประโยชน์**: แยกความรับผิดชอบ, ง่ายต่อการเปลี่ยนแปลง data source หรือ business logic
- **ตัวอย่าง**:

```
interface UserRepository {
    public function find($id);
}

class DbUserRepository implements UserRepository {
    public function find($id) {
        // query DB
    }
}

class UserService {
    private $repo;
    public function __construct(UserRepository $repo) {
        $this->repo = $repo;
    }
    public function getUser($id) {
        return $this->repo->find($id);
```

```
    }
}
```

---

7. Event-driven Programming และ Middleware

- **Event-driven**: การเขียนโปรแกรมที่ตอบสนองต่อเหตุการณ์ (events) เช่น user actions, system events
- **Middleware**: กลไกที่ประมวลผลคำขอก่อนถึงตัว handler (ในเว็บเฟรมเวิร์ก) เช่น การตรวจสอบสิทธิ์, logging
- **ตัวอย่าง Middleware (Laravel)**:

```
public function handle($request, Closure $next)
{
    // ตรวจสอบสิทธิ์
    if (!$request->user()) {
        return redirect('login');
    }
    return $next($request);
}
```

แนวคิด Design Patterns ใน PHP (เชิงลึก)

---

1. Singleton Pattern

ความหมายและปัญหาที่แก้ไข

- Singleton คือ pattern ที่รับประกันว่า คลาสจะมีแค่ instance เดียวตลอดโปรแกรม
- เหมาะกับ resource ที่ต้องแชร์ เช่น การเชื่อมต่อฐานข้อมูล, config manager
- ป้องกันการสร้างอ็อบเจ็กต์ซ้ำซ้อนที่เปลืองหน่วยความจำหรือทำให้สถานะข้อมูลไม่สอดคล้องกัน

หลักการทำงาน

- กำหนด constructor ให้เป็น private เพื่อป้องกันการสร้างอินสแตนซ์จากภายนอก
- สร้าง method getInstance() เพื่อสร้างหรือคืน instance ที่มีอยู่แล้ว

ตัวอย่างโค้ด

```
class DatabaseConnection {
    private static $instance = null;
    private $connection;
```

```
    private function __construct() {
        $this->connection = new PDO('mysql:host=localhost;dbname=testdb', 'user', 'pass');
    }

    public static function getInstance() {
        if (self::$instance === null) {
            self::$instance = new DatabaseConnection();
        }
        return self::$instance;
    }

    public function getConnection() {
        return $this->connection;
    }
}
```

การใช้งาน

```
$db1 = DatabaseConnection::getInstance();
$db2 = DatabaseConnection::getInstance();

var_dump($db1 === $db2); // true
```

---

2. Factory Pattern

ความหมายและปัญหาที่แก้ไข

- Factory Pattern ช่วยแยกการสร้างวัตถุออกจากการใช้งาน ทำให้ระบบยืดหยุ่น
- เหมาะสำหรับระบบที่มีชนิดวัตถุหลายแบบ และอาจเปลี่ยนแปลงได้ง่าย
- ลดการผูกมัด (tight coupling) ระหว่างโค้ดกับชนิดคลาสโดยตรง

หลักการทำงาน

- มีคลาส Factory ที่รับพารามิเตอร์ เช่น ชนิดวัตถุ
- Factory สร้างและคืนอินสแตนซ์วัตถุตามชนิดที่ร้องขอ

ตัวอย่างโค้ด

```
interface Logger {
    public function log(string $message);
}
```

```
class FileLogger implements Logger {
    public function log(string $message) {
        file_put_contents('log.txt', $message.PHP_EOL, FILE_APPEND);
    }
}

class DatabaseLogger implements Logger {
    public function log(string $message) {
        // บันทึกลงฐานข้อมูล
    }
}

class LoggerFactory {
    public static function create(string $type): Logger {
        switch ($type) {
            case 'file':
                return new FileLogger();
            case 'db':
                return new DatabaseLogger();
            default:
                throw new Exception("Unknown logger type");
        }
    }
}
```

การใช้งาน

```
$logger = LoggerFactory::create('file');
$logger->log('This is a log message');
```

---

3. Strategy Pattern

ความหมายและปัญหาที่แก้ไข

- Strategy คือการแยกพฤติกรรมที่เปลี่ยนแปลงได้ออกจากคลาสหลัก
- ช่วยให้เปลี่ยนวิธีการทำงาน (algorithm) ได้โดยไม่ต้องแก้โค้ดที่ใช้งาน

หลักการทำงาน

- สร้าง interface สำหรับพฤติกรรม

- คลาสแต่ละตัวแสดงพฤติกรรมแบบต่าง ๆ
- คลาสหลักรับ strategy มาใช้งาน

ตัวอย่างโค้ด

```
interface CompressionStrategy {
    public function compress(string $data): string;
}

class ZipCompression implements CompressionStrategy {
    public function compress(string $data): string {
        return "ZIP:" . $data; // สมมติการบีบอัด
    }
}

class RarCompression implements CompressionStrategy {
    public function compress(string $data): string {
        return "RAR:" . $data;
    }
}

class CompressionContext {
    private $strategy;

    public function setStrategy(CompressionStrategy $strategy) {
        $this->strategy = $strategy;
    }

    public function compressData(string $data) {
        return $this->strategy->compress($data);
    }
}
```

การใช้งาน

```
$context = new CompressionContext();
$context->setStrategy(new ZipCompression());
echo $context->compressData('Example data'); // ZIP:Example data
```

```
$context->setStrategy(new RarCompression());
echo $context->compressData('Example data'); // RAR:Example data
```

---

4. Observer Pattern

ความหมายและปัญหาที่แก้ไข

- Observer ช่วยให้หลายวัตถุติดตามและตอบสนองเหตุการณ์ที่เกิดในวัตถุหนึ่ง (Subject)
- ช่วยทำให้ระบบยืดหยุ่น ไม่ผูกมัดระหว่าง observer กับ subject

หลักการทำงาน

- Subject มีลิสต์ observer
- Subject แจ้ง observer ทุกครั้งที่สถานะเปลี่ยน
- Observer ตอบสนองด้วย method update()

ตัวอย่างโค้ด

```
interface Observer {
    public function update(string $event);
}

class UserObserver implements Observer {
    public function update(string $event) {
        echo "UserObserver notified of event: $event\n";
    }
}

class Subject {
    private $observers = [];

    public function attach(Observer $obs) {
        $this->observers[] = $obs;
    }

    public function notify(string $event) {
        foreach ($this->observers as $obs) {
            $obs->update($event);
        }
```

```
    }
}
```

การใช้งาน

```
$subject = new Subject();
$subject->attach(new UserObserver());

$subject->notify('UserCreated');
```

---

5. Dependency Injection (DI) และ Inversion of Control (IoC)

ความหมายและปัญหาที่แก้ไข

- DI คือเทคนิคที่ส่ง dependencies (วัตถุที่ต้องใช้งาน) เข้ามายังคลาสภายนอก แทนการสร้างภายใน
- IoC คือหลักการ “กลับด้านการควบคุม” (control) จากคลาสหลักไปให้ external framework หรือ container ดูแล
- ช่วยเพิ่มความยืดหยุ่นและลด coupling ระหว่างคลาส

รูปแบบ DI

- Constructor Injection: ส่ง dependencies ผ่าน constructor
- Setter Injection: ส่งผ่าน method setter
- Interface Injection: ส่งผ่าน interface (ไม่ค่อยใช้ใน PHP)

ตัวอย่างโค้ด

```
class Mailer {
    public function send($message) {
        echo "Sending: $message\n";
    }
}

class UserService {
    private $mailer;

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

    public function notifyUser() {
```

```
        $this->mailer->send("Welcome new user!");
    }
}
```

การใช้งาน

```
$mailer = new Mailer();
$userService = new UserService($mailer);
$userService->notifyUser();
```

---

6. Repository Pattern และ Service Layer

ความหมายและปัญหาที่แก้ไข

- Repository แยกชั้นจัดการข้อมูล (DB, API, cache) ออกจาก business logic
- Service Layer คือชั้นที่รวบรวม business logic และประสานงาน repository
- ช่วยให้โค้ดสะอาด แยกส่วนง่าย ทดสอบง่าย

ตัวอย่างโค้ด

```
interface UserRepository {
    public function find($id);
}

class EloquentUserRepository implements UserRepository {
    public function find($id) {
        return User::find($id); // Eloquent model
    }
}

class UserService {
    private $repo;
    public function __construct(UserRepository $repo) {
        $this->repo = $repo;
    }
    public function getUser($id) {
        return $this->repo->find($id);
    }
}
```

---

7. Event-driven Programming และ Middleware

Event-driven Programming

- โปรแกรมตอบสนองต่อเหตุการณ์ (event) เช่น การกดปุ่ม, การเปลี่ยนสถานะ
- ลดการเขียนโค้ดที่ผูกมัดแบบเรียงลำดับ ทำให้โค้ดยืดหยุ่น

Middleware

- กลไกที่อยู่ระหว่างคำขอ (request) และการตอบกลับ (response)
- ใช้สำหรับตรวจสอบสิทธิ์, logging, แก้ไข request/response
- Framework เช่น Laravel ใช้ middleware อย่างกว้างขวาง

ตัวอย่าง Middleware ใน Laravel

```
public function handle($request, Closure $next)
{
    if (!$request->user()) {
        return redirect('/login');
    }
    return $next($request);
}
```

รายละเอียดเชิงลึก 4 Design Patterns

---

1. Singleton Pattern

แนวคิด

- จำกัดให้คลาสมีแค่ instance เดียวในระบบ
- เหมาะกับ resource ที่ใช้ร่วม เช่น การเชื่อมต่อฐานข้อมูล, logger
- ป้องกันการสร้างวัตถุซ้ำซ้อน

ตัวอย่าง

```
class Singleton {
    private static $instance = null;
    private function __construct() {}
    private function __clone() {}
    public static function getInstance() {
        if (self::$instance === null) {
            self::$instance = new Singleton();
        }
        return self::$instance;
```

```
    }
}

// ใช้งาน
$obj1 = Singleton::getInstance();
$obj2 = Singleton::getInstance();

var_dump($obj1 === $obj2); // true
```

---

2. Factory Pattern

แนวคิด

- แยกการสร้างวัตถุออกจากการใช้งาน
- เพิ่มความยืดหยุ่นและลด coupling

ตัวอย่าง

```
interface Product {
    public function getName();
}

class ProductA implements Product {
    public function getName() {
        return "Product A";
    }
}

class ProductB implements Product {
    public function getName() {
        return "Product B";
    }
}

class ProductFactory {
    public static function create($type) {
        switch($type) {
            case 'A':
```

```
            return new ProductA();
        case 'B':
            return new ProductB();
        default:
            throw new Exception("Unknown product type");
    }
  }
}

// ใช้งาน
$product = ProductFactory::create('A');
echo $product->getName();  // Product A
```

---

3. Strategy Pattern

แนวคิด

- แยกพฤติกรรมออกเป็นคลาสแยกตามกลยุทธ์ (strategy)
- สามารถเปลี่ยนพฤติกรรมได้ตอน runtime

ตัวอย่าง

```
interface PaymentStrategy {
    public function pay($amount);
}

class CreditCardPayment implements PaymentStrategy {
    public function pay($amount) {
        echo "Paid $amount with Credit Card";
    }
}

class PaypalPayment implements PaymentStrategy {
    public function pay($amount) {
        echo "Paid $amount with PayPal";
    }
}
```

```
class ShoppingCart {
    private $paymentMethod;

    public function setPaymentMethod(PaymentStrategy $strategy) {
        $this->paymentMethod = $strategy;
    }

    public function checkout($amount) {
        $this->paymentMethod->pay($amount);
    }
}

// ใช้งาน
$cart = new ShoppingCart();
$cart->setPaymentMethod(new CreditCardPayment());
$cart->checkout(100);  // Paid 100 with Credit Card

$cart->setPaymentMethod(new PaypalPayment());
$cart->checkout(200);  // Paid 200 with PayPal
```

---

4. Observer Pattern

แนวคิด

- กำหนดความสัมพันธ์แบบ 1 ต่อ หลาย (one-to-many)
- เมื่อ subject มีการเปลี่ยนแปลง จะส่ง notification ไปยัง observer หลายตัว

ตัวอย่าง

```
interface Observer {
    public function update($event);
}

class EmailNotifier implements Observer {
    public function update($event) {
        echo "EmailNotifier: New event - $event\n";
    }
}
```

```
class SmsNotifier implements Observer {
    public function update($event) {
        echo "SmsNotifier: New event - $event\n";
    }
}

class Subject {
    private $observers = [];

    public function attach(Observer $obs) {
        $this->observers[] = $obs;
    }

    public function notify($event) {
        foreach ($this->observers as $obs) {
            $obs->update($event);
        }
    }

    public function doSomething() {
        // ทำบางอย่าง แล้วแจ้ง observer
        $this->notify('Something happened');
    }
}

// ใช้งาน
$subject = new Subject();
$subject->attach(new EmailNotifier());
$subject->attach(new SmsNotifier());

$subject->doSomething();

// Output:
```

```
// EmailNotifier: New event - Something happened
// SmsNotifier: New event - Something happened
```

นี่คือตัวอย่างโปรแกรม PHP แบบเต็มไฟล์ ครอบคลุม 4 Design Patterns (Singleton, Factory, Strategy, Observer) ทั้งแบบพื้นฐานและแนวประยุกต์ จำนวนรวม 6 โปรแกรม พร้อมโครงสร้างและคำอธิบาย

ตัวอย่างโปรแกรมพื้นฐาน 3 โปรแกรม

ตัวอย่างที่ 1: Singleton Pattern (Database Connection)

โครงสร้างไฟล์

/singleton/

Database.php

index.php

Database.php

```
<?php
class Database {
    private static $instance = null;
    private $connection;

    private function __construct() {
        // ตัวอย่างเชื่อมต่อ PDO จริงๆต้องใส่ host/db/user/pass
        $this->connection = new PDO('sqlite::memory:');
    }

    public static function getInstance() {
        if (self::$instance === null) {
            self::$instance = new Database();
        }
        return self::$instance;
    }

    public function getConnection() {
```

```
        return $this->connection;
    }
}
```

---

index.php

```
<?php
require_once 'Database.php';

$db1 = Database::getInstance();
$db2 = Database::getInstance();

if ($db1 === $db2) {
    echo "Singleton works: Both instances are the same.\n";
} else {
    echo "Singleton failed: Instances are different.\n";
}
```

คำอธิบาย

- Database class มี constructor เป็น private
- getInstance() คืน instance เดียวกันเสมอ
- ทดสอบโดยเปรียบเทียบสองตัวแปร ถ้าเหมือนกันแสดงว่า Singleton ทำงานถูกต้อง

ผลการรัน

```
Singleton works: Both instances are the same.
```

ตัวอย่างที่ 2: Factory Pattern (สร้างสัตว์เลี้ยง)

โครงสร้างไฟล์

```
/factory/
    Animal.php
    Dog.php
    Cat.php
    AnimalFactory.php
    index.php
```

Animal.php

```
<?php
interface Animal {
    public function speak();
}
```

Dog.php

```
<?php
require_once 'Animal.php';

class Dog implements Animal {
    public function speak() {
        return "Woof!";
    }
}
```

Cat.php

```
<?php
require_once 'Animal.php';

class Cat implements Animal {
    public function speak() {
        return "Meow!";
    }
}
```

AnimalFactory.php

```
<?php
require_once 'Dog.php';
require_once 'Cat.php';

class AnimalFactory {
    public static function create($type) {
        switch(strtolower($type)) {
```

```
            case 'dog':
                return new Dog();
            case 'cat':
                return new Cat();
            default:
                throw new Exception("Unknown animal type");
        }
    }
}
```

---

index.php

```
<?php
require_once 'AnimalFactory.php';

try {
    $dog = AnimalFactory::create('dog');
    echo $dog->speak() . "\n";  // Woof!

    $cat = AnimalFactory::create('cat');
    echo $cat->speak() . "\n";  // Meow!
} catch (Exception $e) {
    echo "Error: " . $e->getMessage();
}
```

คำอธิบาย

- Animal interface กำหนด method speak()
- Dog, Cat implement Animal
- AnimalFactory สร้าง instance ตามชนิดที่ขอ
- index.php ทดสอบสร้างและเรียก speak()

ผลการรัน

```
Woof!
Meow!
```

ตัวอย่างที่ 3: Strategy Pattern (ชำระเงิน)

โครงสร้างไฟล์

/strategy/

PaymentStrategy.php

CreditCardPayment.php

PaypalPayment.php

ShoppingCart.php

index.php

PaymentStrategy.php

```
<?php
interface PaymentStrategy {
    public function pay($amount);
}
```

CreditCardPayment.php

```
<?php
require_once 'PaymentStrategy.php';

class CreditCardPayment implements PaymentStrategy {
    public function pay($amount) {
        echo "Paying $amount using Credit Card.\n";
    }
}
```

PaypalPayment.php

```
<?php
require_once 'PaymentStrategy.php';

class PaypalPayment implements PaymentStrategy {
    public function pay($amount) {
        echo "Paying $amount using PayPal.\n";
    }
}
```

ShoppingCart.php

```
<?php
require_once 'PaymentStrategy.php';

class ShoppingCart {
    private $paymentMethod;

    public function setPaymentMethod(PaymentStrategy $method) {
        $this->paymentMethod = $method;
    }

    public function checkout($amount) {
        $this->paymentMethod->pay($amount);
    }
}
```

index.php

```
<?php
require_once 'CreditCardPayment.php';
require_once 'PaypalPayment.php';
require_once 'ShoppingCart.php';

$cart = new ShoppingCart();

$cart->setPaymentMethod(new CreditCardPayment());
$cart->checkout(150); // Paying 150 using Credit Card.

$cart->setPaymentMethod(new PaypalPayment());
$cart->checkout(250); // Paying 250 using PayPal.
```

คำอธิบาย

- สร้าง interface PaymentStrategy
- สอง strategy: CreditCardPayment, PaypalPayment

- ShoppingCart เลือก strategy runtime แล้วจ่ายเงิน
- index.php ทดสอบเปลี่ยนวิธีจ่ายเงิน

ผลการรัน

Paying 150 using Credit Card.

Paying 250 using PayPal.

ตัวอย่างโปรแกรมแนวประยุกต์ 3 โปรแกรม

ตัวอย่างที่ 4: Observer Pattern (ระบบแจ้งเตือนผู้ใช้)

โครงสร้างไฟล์

/observer/

Observer.php

UserNotifier.php

AdminNotifier.php

Subject.php

UserRegistration.php

index.php

Observer.php

```
<?php
interface Observer {
    public function update($event);
}
```

UserNotifier.php

```
<?php
require_once 'Observer.php';

class UserNotifier implements Observer {
    public function update($event) {
        echo "User notified: $event\n";
    }
}
```

AdminNotifier.php

```
<?php
require_once 'Observer.php';

class AdminNotifier implements Observer {
    public function update($event) {
        echo "Admin notified: $event\n";
    }
}
```

Subject.php

```
<?php
require_once 'Observer.php';

class Subject {
    private $observers = [];

    public function attach(Observer $observer) {
        $this->observers[] = $observer;
    }

    public function notify($event) {
        foreach ($this->observers as $obs) {
            $obs->update($event);
        }
    }
}
```

UserRegistration.php

```
<?php
require_once 'Subject.php';

class UserRegistration extends Subject {
```

```
    public function register($username) {
        // สมมติสร้างผู้ใช้ในระบบ
        echo "User $username registered successfully.\n";
        $this->notify("New user registered: $username");
    }
}
```

---

index.php

```
<?php
require_once 'UserNotifier.php';
require_once 'AdminNotifier.php';
require_once 'UserRegistration.php';

$userNotifier = new UserNotifier();
$adminNotifier = new AdminNotifier();

$registration = new UserRegistration();
$registration->attach($userNotifier);
$registration->attach($adminNotifier);

$registration->register('john_doe');
```

คำอธิบาย

- Observer interface
- UserNotifier และ AdminNotifier รับการแจ้งเตือน
- Subject เก็บ observer และแจ้งเมื่อเหตุการณ์เกิดขึ้น
- UserRegistration สืบทอด Subject และแจ้งเหตุการณ์เมื่อมีผู้ลงทะเบียน
- ทดสอบการแจ้งเตือนผ่าน index.php

ผลการรัน

```
User john_doe registered successfully.
User notified: New user registered: john_doe
Admin notified: New user registered: john_doe
```

ตัวอย่างที่ 5: Factory Pattern + Strategy Pattern ผสม (ระบบแจ้งเตือนแบบเลือกช่องทาง)

โครงสร้างไฟล์

/notification-system/

NotificationStrategy.php

EmailNotification.php

SmsNotification.php

NotificationFactory.php

NotificationService.php

index.php

NotificationStrategy.php

```
<?php
interface NotificationStrategy {
    public function notify($message);
}
```

EmailNotification.php

```
<?php
require_once 'NotificationStrategy.php';

class EmailNotification implements NotificationStrategy {
    public function notify($message) {
        echo "Send Email: $message\n";
    }
}
```

SmsNotification.php

```
<?php
require_once 'NotificationStrategy.php';

class SmsNotification implements NotificationStrategy {
    public function notify($message) {
        echo "Send SMS: $message\n";
    }
```

```
}
```

NotificationFactory.php

```
<?php
require_once 'EmailNotification.php';
require_once 'SmsNotification.php';

class NotificationFactory {
    public static function create($type) {
        switch(strtolower($type)) {
            case 'email': return new EmailNotification();
            case 'sms': return new SmsNotification();
            default: throw new Exception("Unknown notification type");
        }
    }
}
```

NotificationService.php

```
<?php
class NotificationService {
    private $strategy;

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

    public function send($message) {
        $this->strategy->notify($message);
    }
}
```

index.php

```
<?php
require_once 'NotificationFactory.php';
```

```
require_once 'NotificationService.php';

try {
    $emailStrategy = NotificationFactory::create('email');
    $smsStrategy = NotificationFactory::create('sms');

    $service = new NotificationService($emailStrategy);
    $service->send('Welcome to our service!');

    $service = new NotificationService($smsStrategy);
    $service->send('Your order has shipped!');
} catch (Exception $e) {
    echo "Error: " . $e->getMessage();
}
```

คำอธิบาย

- Factory สร้าง strategy ตามประเภท notification
- NotificationService ใช้ strategy ส่งข้อความ
- index.php ทดสอบส่งข้อความผ่าน Email และ SMS

ผลการรัน

```
Send Email: Welcome to our service!
Send SMS: Your order has shipped!
```

ตัวอย่างที่ 6: Singleton + Observer (Logger แบบ Singleton แจ้ง Observer)

โครงสร้างไฟล์

```
/logger-observer/
    Logger.php
    LoggerObserver.php
    index.php
```

Logger.php

```
<?php
require_once 'LoggerObserver.php';
```

```
class Logger {
    private static $instance = null;
    private $observers = [];

    private function __construct() {}

    public static function getInstance() {
        if (self::$instance === null) {
            self::$instance = new Logger();
        }
        return self::$instance;
    }

    public function attach(LoggerObserver $observer) {
        $this->observers[] = $observer;
    }

    public function log($message) {
        echo "Logging: $message\n";
        $this->notify($message);
    }

    private function notify($message) {
        foreach ($this->observers as $obs) {
            $obs->update($message);
        }
    }
}
```

LoggerObserver.php

```
<?php
class LoggerObserver {
    public function update($message) {
```

```
        echo "Observer received log: $message\n";
    }
}
```

---

index.php

```
<?php
require_once 'Logger.php';
require_once 'LoggerObserver.php';

$logger = Logger::getInstance();

$observer1 = new LoggerObserver();
$observer2 = new LoggerObserver();

$logger->attach($observer1);
$logger->attach($observer2);

$logger->log("User logged in");
```

คำอธิบาย

- Logger เป็น Singleton เก็บ observer
- เวลาบันทึก log จะ notify observer ทั้งหมด
- index.php ทดสอบใช้งาน Logger และ observer

ผลการรัน

```
Logging: User logged in
Observer received log: User logged in
Observer received log: User logged in
```

Dependency Injection (DI) และ Inversion of Control (IoC) ใน PHP

แนวคิดหลัก