Pertemuan 13: Prinsip SOLID

Tujuan Pembelajaran

Setelah mengikuti pertemuan ini, mahasiswa diharapkan dapat:

1. Memahami 5 prinsip SOLID dalam Object-Oriented Design
2. Mengidentifikasi violation dari prinsip SOLID dalam code
3. Menerapkan Single Responsibility Principle (SRP)
4. Mengimplementasikan Open/Closed Principle (OCP)
5. Menguasai Liskov Substitution Principle (LSP)
6. Menerapkan Interface Segregation Principle (ISP)
7. Menggunakan Dependency Inversion Principle (DIP)
8. Merefactor code untuk compliance dengan SOLID principles

Pengenalan SOLID Principles

Definisi SOLID

SOLID adalah akronim dari 5 prinsip fundamental dalam Object-Oriented Design yang dikembangkan oleh Robert C. Martin (Uncle Bob). Prinsip-prinsip ini membantu developer membuat software yang:

- **Maintainable** - Mudah dipelihara dan dimodifikasi
- **Extensible** - Mudah ditambahkan fitur baru
- **Testable** - Mudah untuk ditest
- **Readable** - Code yang mudah dipahami
- **Flexible** - Adaptable terhadap perubahan requirements

SOLID Acronym

- **S** - Single Responsibility Principle (SRP)
- **O** - Open/Closed Principle (OCP)
- **L** - Liskov Substitution Principle (LSP)
- **I** - Interface Segregation Principle (ISP)
- **D** - Dependency Inversion Principle (DIP)

Manfaat Menerapkan SOLID

1. **Reduced Coupling** - Komponen loosely coupled
2. **Increased Cohesion** - Komponen yang related berada bersama
3. **Better Testability** - Easier unit testing
4. **Improved Maintainability** - Easier to modify dan extend
5. **Enhanced Readability** - Code yang self-documenting

1. Single Responsibility Principle (SRP)

Definisi

> "A class should have only one reason to change"

Setiap class harus memiliki hanya satu tanggung jawab dan hanya satu alasan untuk berubah.

Contoh Violation SRP

```php
<?php
// ❌ BAD - Multiple responsibilities
class User {
    private $name;
    private $email;

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

    // Responsibility 1: User data management
    public function getName() { return $this->name; }
    public function getEmail() { return $this->email; }

    // Responsibility 2: Database operations
    public function save() {
        // Database logic
        $sql = "INSERT INTO users (name, email) VALUES (?, ?)";
        // Execute query...
    }

    // Responsibility 3: Email operations
    public function sendWelcomeEmail() {
        // Email logic
        mail($this->email, "Welcome", "Welcome to our site!");
    }

    // Responsibility 4: Validation
    public function validate() {
        if (empty($this->name)) throw new Exception("Name required");
        if (!filter_var($this->email, FILTER_VALIDATE_EMAIL)) {
            throw new Exception("Invalid email");
        }
    }
}
```

Contoh Penerapan SRP

```php
<?php
// ✅ GOOD - Single responsibility per class

// Responsibility 1: User data representation
class User {
```

```
    private $name;
    private $email;

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

    public function getName() { return $this->name; }
    public function getEmail() { return $this->email; }
}

// Responsibility 2: User validation
class UserValidator {
    public function validate(User $user) {
        if (empty($user->getName())) {
            throw new ValidationException("Name is required");
        }

        if (!filter_var($user->getEmail(), FILTER_VALIDATE_EMAIL)) {
            throw new ValidationException("Invalid email format");
        }
    }
}

// Responsibility 3: Database operations
class UserRepository {
    private $database;

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

    public function save(User $user) {
        $sql = "INSERT INTO users (name, email) VALUES (?, ?)";
        $this->database->execute($sql, [$user->getName(), $user-
>getEmail()]);
    }

    public function findByEmail($email) {
        $sql = "SELECT * FROM users WHERE email = ?";
        return $this->database->fetch($sql, [$email]);
    }
}

// Responsibility 4: Email services
class EmailService {
    public function sendWelcomeEmail(User $user) {
        $subject = "Welcome to Our Platform";
        $message = "Hello {$user->getName()}, welcome to our platform!";

        mail($user->getEmail(), $subject, $message);
    }
}
```

2. Open/Closed Principle (OCP)

Definisi

> "Software entities should be open for extension but closed for modification"

Class harus dapat diperluas (extended) tanpa memodifikasi source code yang sudah ada.

Contoh Violation OCP

```php
<?php
// ❌ BAD - Modifying existing code for new features
class DiscountCalculator {
    public function calculateDiscount($customerType, $amount) {
        switch ($customerType) {
            case 'regular':
                return $amount * 0.05;
            case 'premium':
                return $amount * 0.10;
            case 'vip':
                return $amount * 0.15;
            default:
                return 0;
        }
    }
}

// Adding new customer type requires modifying existing code
// What if we want to add 'corporate' customer type?
```

Contoh Penerapan OCP

```php
<?php
// ✅ GOOD - Using abstraction for extension

interface DiscountStrategyInterface {
    public function calculateDiscount($amount);
    public function getCustomerType();
}

class RegularCustomerDiscount implements DiscountStrategyInterface {
    public function calculateDiscount($amount) {
        return $amount * 0.05;
    }

    public function getCustomerType() {
        return 'regular';
    }
```

```
}

class PremiumCustomerDiscount implements DiscountStrategyInterface {
    public function calculateDiscount($amount) {
        return $amount * 0.10;
    }

    public function getCustomerType() {
        return 'premium';
    }
}

class VipCustomerDiscount implements DiscountStrategyInterface {
    public function calculateDiscount($amount) {
        return $amount * 0.15;
    }

    public function getCustomerType() {
        return 'vip';
    }
}

// New customer type can be added without modifying existing code
class CorporateCustomerDiscount implements DiscountStrategyInterface {
    public function calculateDiscount($amount) {
        return $amount * 0.20;
    }

    public function getCustomerType() {
        return 'corporate';
    }
}

class DiscountCalculator {
    private $strategies = [];

    public function addStrategy(DiscountStrategyInterface $strategy) {
        $this->strategies[$strategy->getCustomerType()] = $strategy;
    }

    public function calculateDiscount($customerType, $amount) {
        if (isset($this->strategies[$customerType])) {
            return $this->strategies[$customerType]-
>calculateDiscount($amount);
        }

        return 0;
    }
}
```

3. Liskov Substitution Principle (LSP)

Definisi

> "Objects of a superclass should be replaceable with objects of a subclass without breaking the application"

Subclass harus dapat menggantikan parent class tanpa mengubah functionality program.

Contoh Violation LSP

```php
<?php
// ❌ BAD - Subclass changes expected behavior

class Rectangle {
    protected $width;
    protected $height;

    public function setWidth($width) {
        $this->width = $width;
    }

    public function setHeight($height) {
        $this->height = $height;
    }

    public function getArea() {
        return $this->width * $this->height;
    }
}

class Square extends Rectangle {
    public function setWidth($width) {
        $this->width = $width;
        $this->height = $width; // Violating LSP - changing behavior
    }

    public function setHeight($height) {
        $this->height = $height;
        $this->width = $height; // Violating LSP - changing behavior
    }
}

// This will fail for Square objects
function testRectangle(Rectangle $rectangle) {
    $rectangle->setWidth(10);
    $rectangle->setHeight(5);

    // Expected: 50, but Square will return 25
    assert($rectangle->getArea() === 50); // Fails for Square
}
```

Contoh Penerapan LSP

```php
<?php
// ✅ GOOD - Proper abstraction respecting LSP

interface ShapeInterface {
    public function getArea();
}

class Rectangle implements ShapeInterface {
    private $width;
    private $height;

    public function __construct($width, $height) {
        $this->width = $width;
        $this->height = $height;
    }

    public function getArea() {
        return $this->width * $this->height;
    }

    public function setWidth($width) {
        $this->width = $width;
    }

    public function setHeight($height) {
        $this->height = $height;
    }
}

class Square implements ShapeInterface {
    private $side;

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

    public function getArea() {
        return $this->side * $this->side;
    }

    public function setSide($side) {
        $this->side = $side;
    }
}

// Both can be used interchangeably when treated as ShapeInterface
function calculateTotalArea(array $shapes) {
    $totalArea = 0;
    foreach ($shapes as $shape) {
        if ($shape instanceof ShapeInterface) {
            $totalArea += $shape->getArea();
        }
```

```
        }
        return $totalArea;
    }
```

4. Interface Segregation Principle (ISP)

Definisi

> "No client should be forced to depend on methods it does not use"

Interface harus specific dan focused, tidak memaksa implementor untuk mengimplementasikan method yang tidak dibutuhkan.

Contoh Violation ISP

```php
<?php
// ❌ BAD - Fat interface forcing unused methods

interface WorkerInterface {
    public function work();
    public function eat();
    public function sleep();
}

class HumanWorker implements WorkerInterface {
    public function work() {
        echo "Human working";
    }

    public function eat() {
        echo "Human eating";
    }

    public function sleep() {
        echo "Human sleeping";
    }
}

class RobotWorker implements WorkerInterface {
    public function work() {
        echo "Robot working";
    }

    // Forced to implement methods that robots don't need
    public function eat() {
        throw new Exception("Robots don't eat");
    }

    public function sleep() {
        throw new Exception("Robots don't sleep");
```

```
    }
}
```

Contoh Penerapan ISP

```php
<?php
// ✅ GOOD - Segregated interfaces

interface WorkableInterface {
    public function work();
}

interface EatableInterface {
    public function eat();
}

interface SleepableInterface {
    public function sleep();
}

class HumanWorker implements WorkableInterface, EatableInterface,
SleepableInterface {
    public function work() {
        echo "Human working";
    }

    public function eat() {
        echo "Human eating";
    }

    public function sleep() {
        echo "Human sleeping";
    }
}

class RobotWorker implements WorkableInterface {
    public function work() {
        echo "Robot working";
    }

    // Only implements what it needs
}

class WorkManager {
    public function manageWork(WorkableInterface $worker) {
        $worker->work();
    }

    public function manageMealTime(EatableInterface $worker) {
        $worker->eat();
    }
```

```
    public function manageSleepTime(SleepableInterface $worker) {
        $worker->sleep();
    }
}
```

5. Dependency Inversion Principle (DIP)

Definisi

> "High-level modules should not depend on low-level modules. Both should depend on abstractions"

Dependency harus pada abstraction (interface), bukan pada concrete implementation.

Contoh Violation DIP

```
<?php
// ❌ BAD - Direct dependency on concrete classes

class MySQLDatabase {
    public function save($data) {
        // MySQL specific implementation
        echo "Saving to MySQL: " . json_encode($data);
    }
}

class EmailService {
    public function send($to, $subject, $message) {
        // Direct email implementation
        echo "Sending email to {$to}: {$subject}";
    }
}

class UserService {
    private $database;
    private $emailService;

    public function __construct() {
        // Direct dependency on concrete classes
        $this->database = new MySQLDatabase();
        $this->emailService = new EmailService();
    }

    public function registerUser($userData) {
        $this->database->save($userData);
        $this->emailService->send($userData['email'], 'Welcome',
'Welcome!');
    }
}

// Hard to test, hard to change database or email provider
```

Contoh Penerapan DIP

```php
<?php
// ✅ GOOD - Dependency on abstractions

interface DatabaseInterface {
    public function save($data);
    public function find($id);
}

interface EmailServiceInterface {
    public function send($to, $subject, $message);
}

class MySQLDatabase implements DatabaseInterface {
    public function save($data) {
        echo "Saving to MySQL: " . json_encode($data);
    }

    public function find($id) {
        echo "Finding in MySQL: {$id}";
        return ['id' => $id, 'data' => 'sample'];
    }
}

class PostgreSQLDatabase implements DatabaseInterface {
    public function save($data) {
        echo "Saving to PostgreSQL: " . json_encode($data);
    }

    public function find($id) {
        echo "Finding in PostgreSQL: {$id}";
        return ['id' => $id, 'data' => 'sample'];
    }
}

class SMTPEmailService implements EmailServiceInterface {
    public function send($to, $subject, $message) {
        echo "Sending via SMTP to {$to}: {$subject}";
    }
}

class SendGridEmailService implements EmailServiceInterface {
    public function send($to, $subject, $message) {
        echo "Sending via SendGrid to {$to}: {$subject}";
    }
}

class UserService {
    private $database;
    private $emailService;
```

```
    // Dependency injection - depends on abstractions
    public function __construct(DatabaseInterface $database,
EmailServiceInterface $emailService) {
        $this->database = $database;
        $this->emailService = $emailService;
    }

    public function registerUser($userData) {
        $this->database->save($userData);
        $this->emailService->send($userData['email'], 'Welcome', 'Welcome
to our platform!');
    }
}

// Easy to test with mock objects, easy to swap implementations
$userService = new UserService(
    new MySQLDatabase(),
    new SMTPEmailService()
);

// Or switch to different implementations
$userService = new UserService(
    new PostgreSQLDatabase(),
    new SendGridEmailService()
);
```

SOLID dalam Practice: E-commerce Example

Struktur Aplikasi E-commerce dengan SOLID

```
<?php
// Domain Models (SRP)
class Product {
    private $id;
    private $name;
    private $price;
    private $category;

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

    // Getters only - single responsibility
    public function getId() { return $this->id; }
    public function getName() { return $this->name; }
    public function getPrice() { return $this->price; }
    public function getCategory() { return $this->category; }
```

```php
}

class Order {
    private $id;
    private $customerId;
    private $items = [];
    private $status;

    public function __construct($id, $customerId) {
        $this->id = $id;
        $this->customerId = $customerId;
        $this->status = 'pending';
    }

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

    public function getTotal() {
        return array_sum(array_map(function($item) {
            return $item->getSubtotal();
        }, $this->items));
    }

    // Getters
    public function getId() { return $this->id; }
    public function getCustomerId() { return $this->customerId; }
    public function getItems() { return $this->items; }
    public function getStatus() { return $this->status; }

    public function setStatus($status) {
        $this->status = $status;
    }
}

class OrderItem {
    private $product;
    private $quantity;

    public function __construct(Product $product, $quantity) {
        $this->product = $product;
        $this->quantity = $quantity;
    }

    public function getSubtotal() {
        return $this->product->getPrice() * $this->quantity;
    }

    public function getProduct() { return $this->product; }
    public function getQuantity() { return $this->quantity; }
}

// Repository Interfaces (DIP)
interface ProductRepositoryInterface {
```

```
    public function findById($id);
    public function findByCategory($category);
    public function save(Product $product);
}

interface OrderRepositoryInterface {
    public function findById($id);
    public function save(Order $order);
    public function findByCustomerId($customerId);
}

// Pricing Strategies (OCP)
interface PricingStrategyInterface {
    public function calculatePrice(Order $order);
}

class RegularPricing implements PricingStrategyInterface {
    public function calculatePrice(Order $order) {
        return $order->getTotal();
    }
}

class BulkDiscountPricing implements PricingStrategyInterface {
    private $threshold;
    private $discountPercent;

    public function __construct($threshold = 1000, $discountPercent = 0.1)
{
        $this->threshold = $threshold;
        $this->discountPercent = $discountPercent;
    }

    public function calculatePrice(Order $order) {
        $total = $order->getTotal();

        if ($total >= $this->threshold) {
            return $total * (1 - $this->discountPercent);
        }

        return $total;
    }
}

// Notification Interfaces (ISP)
interface OrderNotificationInterface {
    public function notifyOrderPlaced(Order $order);
}

interface PaymentNotificationInterface {
    public function notifyPaymentReceived(Order $order);
}

interface ShippingNotificationInterface {
    public function notifyOrderShipped(Order $order);
```

```
}

// Services (SRP + DIP)
class OrderService {
    private $orderRepository;
    private $productRepository;
    private $pricingStrategy;
    private $notifications;

    public function __construct(
        OrderRepositoryInterface $orderRepository,
        ProductRepositoryInterface $productRepository,
        PricingStrategyInterface $pricingStrategy,
        array $notifications = []
    ) {
        $this->orderRepository = $orderRepository;
        $this->productRepository = $productRepository;
        $this->pricingStrategy = $pricingStrategy;
        $this->notifications = $notifications;
    }

    public function createOrder($customerId, array $items) {
        $order = new Order(uniqid('order_'), $customerId);

        foreach ($items as $itemData) {
            $product = $this->productRepository-
>findById($itemData['product_id']);
            if ($product) {
                $orderItem = new OrderItem($product,
$itemData['quantity']);
                $order->addItem($orderItem);
            }
        }

        $this->orderRepository->save($order);

        // Notify interested parties
        foreach ($this->notifications as $notification) {
            if ($notification instanceof OrderNotificationInterface) {
                $notification->notifyOrderPlaced($order);
            }
        }

        return $order;
    }

    public function calculateOrderTotal(Order $order) {
        return $this->pricingStrategy->calculatePrice($order);
    }
}

// Notification Implementations
class EmailNotificationService implements
    OrderNotificationInterface,
```

```
    PaymentNotificationInterface,
    ShippingNotificationInterface
{
    private $emailService;

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

    public function notifyOrderPlaced(Order $order) {
        echo "Email: Order {$order->getId()} placed\n";
    }

    public function notifyPaymentReceived(Order $order) {
        echo "Email: Payment received for order {$order->getId()}\n";
    }

    public function notifyOrderShipped(Order $order) {
        echo "Email: Order {$order->getId()} shipped\n";
    }
}

class SMSNotificationService implements OrderNotificationInterface {
    public function notifyOrderPlaced(Order $order) {
        echo "SMS: Order {$order->getId()} placed\n";
    }
}
```

SOLID Checklist

Single Responsibility Principle Checklist

- [ ] Each class has only one reason to change
- [ ] Class names clearly indicate their single responsibility
- [ ] Methods in a class are cohesive and related
- [ ] No mixed concerns (e.g., data access + business logic)

Open/Closed Principle Checklist

- [ ] New functionality can be added without modifying existing code
- [ ] Abstractions (interfaces/abstract classes) are used for extension points
- [ ] Strategy pattern is used for varying algorithms
- [ ] Template method pattern is used for varying steps

Liskov Substitution Principle Checklist

- [ ] Subclasses can replace parent classes without breaking functionality
- [ ] Preconditions are not strengthened in subclasses
- [ ] Postconditions are not weakened in subclasses
- [ ] Method signatures are compatible between parent and child

Interface Segregation Principle Checklist

- [ ] Interfaces are focused and cohesive
- [ ] Clients don't depend on methods they don't use
- [ ] Large interfaces are broken into smaller, specific ones
- [ ] Role-based interfaces are used

Dependency Inversion Principle Checklist

- [ ] High-level modules depend on abstractions, not concretions
- [ ] Dependencies are injected rather than created internally
- [ ] Abstractions don't depend on details
- [ ] Details depend on abstractions

Refactoring untuk SOLID Compliance

Before: Violating Multiple SOLID Principles

```php
<?php
class UserManager {
    private $db;

    public function __construct() {
        $this->db = new PDO('mysql:host=localhost;dbname=app', 'user',
'pass');
    }

    public function createUser($name, $email, $password) {
        // Validation
        if (empty($name)) throw new Exception("Name required");
        if (!filter_var($email, FILTER_VALIDATE_EMAIL)) throw new
Exception("Invalid email");

        // Password hashing
        $hashedPassword = password_hash($password, PASSWORD_DEFAULT);

        // Database operation
        $stmt = $this->db->prepare("INSERT INTO users (name, email,
password) VALUES (?, ?, ?)");
        $stmt->execute([$name, $email, $hashedPassword]);

        // Email sending
        mail($email, "Welcome", "Welcome to our platform!");

        // Logging
        error_log("User created: " . $email);

        return $this->db->lastInsertId();
    }

    public function updateUser($id, $name, $email) {
```

```
        // Similar mixed responsibilities...
    }

    public function deleteUser($id) {
        // Similar mixed responsibilities...
    }
}
```

After: SOLID Compliant

```
<?php
// Separated responsibilities

interface UserRepositoryInterface {
    public function save(User $user);
    public function findById($id);
    public function update(User $user);
    public function delete($id);
}

interface UserValidatorInterface {
    public function validate(User $user);
}

interface EmailServiceInterface {
    public function sendWelcomeEmail(User $user);
}

interface LoggerInterface {
    public function log($message, $context = []);
}

class User {
    private $id;
    private $name;
    private $email;
    private $password;

    // Constructor and getters/setters
}

class UserValidator implements UserValidatorInterface {
    public function validate(User $user) {
        $errors = [];

        if (empty($user->getName())) {
            $errors[] = "Name is required";
        }

        if (!filter_var($user->getEmail(), FILTER_VALIDATE_EMAIL)) {
            $errors[] = "Invalid email format";
```

```
        }

        if (!empty($errors)) {
            throw new ValidationException("Validation failed", $errors);
        }
    }
}

class UserService {
    private $repository;
    private $validator;
    private $emailService;
    private $logger;

    public function __construct(
        UserRepositoryInterface $repository,
        UserValidatorInterface $validator,
        EmailServiceInterface $emailService,
        LoggerInterface $logger
    ) {
        $this->repository = $repository;
        $this->validator = $validator;
        $this->emailService = $emailService;
        $this->logger = $logger;
    }

    public function createUser($name, $email, $password) {
        $user = new User($name, $email, password_hash($password,
PASSWORD_DEFAULT));

        $this->validator->validate($user);

        $userId = $this->repository->save($user);
        $user->setId($userId);

        $this->emailService->sendWelcomeEmail($user);

        $this->logger->log("User created successfully", ['user_id' =>
$userId]);

        return $userId;
    }
}
```

Testing SOLID Code

Unit Testing dengan SOLID

```
<?php
use PHPUnit\Framework\TestCase;
```

```php
class UserServiceTest extends TestCase {
    private $repository;
    private $validator;
    private $emailService;
    private $logger;
    private $userService;

    protected function setUp(): void {
        // Easy to mock dependencies thanks to DIP
        $this->repository = $this->createMock(UserRepositoryInterface::class);
        $this->validator = $this->createMock(UserValidatorInterface::class);
        $this->emailService = $this->createMock(EmailServiceInterface::class);
        $this->logger = $this->createMock(LoggerInterface::class);

        $this->userService = new UserService(
            $this->repository,
            $this->validator,
            $this->emailService,
            $this->logger
        );
    }

    public function testCreateUserSuccess() {
        // Setup expectations
        $this->validator
            ->expects($this->once())
            ->method('validate');

        $this->repository
            ->expects($this->once())
            ->method('save')
            ->willReturn(123);

        $this->emailService
            ->expects($this->once())
            ->method('sendWelcomeEmail');

        $this->logger
            ->expects($this->once())
            ->method('log');

        // Test
        $userId = $this->userService->createUser('John Doe', 'john@example.com', 'password');

        // Assert
        $this->assertEquals(123, $userId);
    }

    public function testCreateUserValidationFailure() {
        $this->validator
```

```
            ->expects($this->once())
            ->method('validate')
            ->willThrowException(new ValidationException("Validation
failed"));

        $this->expectException(ValidationException::class);

        $this->userService->createUser('', 'invalid-email', 'weak');
    }
}
```

Best Practices untuk SOLID

1. Start with Interfaces

```php
<?php
// Define contracts first
interface PaymentProcessorInterface {
    public function processPayment($amount, $paymentMethod);
}

interface InventoryServiceInterface {
    public function checkStock($productId);
    public function reserveStock($productId, $quantity);
}

// Then implement
class StripePaymentProcessor implements PaymentProcessorInterface {
    public function processPayment($amount, $paymentMethod) {
        // Stripe implementation
    }
}
```

2. Use Dependency Injection Container

```php
<?php
class DIContainer {
    private $bindings = [];
    private $instances = [];

    public function bind($abstract, $concrete, $singleton = false) {
        $this->bindings[$abstract] = compact('concrete', 'singleton');
    }

    public function resolve($abstract) {
        if (isset($this->instances[$abstract])) {
            return $this->instances[$abstract];
        }
```

```
        if (!isset($this->bindings[$abstract])) {
            throw new Exception("No binding found for {$abstract}");
        }

        $binding = $this->bindings[$abstract];
        $instance = $this->createInstance($binding['concrete']);

        if ($binding['singleton']) {
            $this->instances[$abstract] = $instance;
        }

        return $instance;
    }

    private function createInstance($concrete) {
        if ($concrete instanceof Closure) {
            return $concrete($this);
        }

        return new $concrete();
    }
}

// Usage
$container = new DIContainer();

$container->bind(UserRepositoryInterface::class, function($container) {
    return new DatabaseUserRepository($container->resolve('database'));
});

$container->bind(EmailServiceInterface::class, SMTPEmailService::class,
true);

$userService = $container->resolve(UserService::class);
```

3. Progressive Refactoring

1. **Start with SRP** - Extract classes with single responsibilities
2. **Add abstractions** - Create interfaces for extension points
3. **Use dependency injection** - Inject dependencies instead of creating them
4. **Split large interfaces** - Create focused interfaces
5. **Follow LSP** - Ensure substitutability

Common SOLID Violations dan Solutions

Fat Controllers

```
<?php
// ❌ BAD
class UserController {
```

```
    public function store(Request $request) {
        // Validation
        if (empty($request->name)) return error('Name required');

        // Business logic
        $user = new User();
        $user->name = $request->name;
        $user->email = $request->email;

        // Database
        DB::table('users')->insert($user->toArray());

        // Email
        Mail::send('welcome', $user, function($m) use ($user) {
            $m->to($user->email)->subject('Welcome');
        });

        return response()->json(['success' => true]);
    }
}

// ✅ GOOD
class UserController {
    private $userService;

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

    public function store(CreateUserRequest $request) {
        $result = $this->userService->createUser($request->validated());

        return response()->json($result);
    }
}
```

Contoh Implementasi

Lihat file `example.php` untuk berbagai contoh implementasi prinsip SOLID dalam PHP OOP.

Latihan

1. Identifikasi SOLID violations dalam legacy code
2. Refactor monolithic class menjadi SOLID compliant
3. Implement dependency injection container
4. Create extensible system menggunakan Strategy pattern

Tugas Rumah

Refactor aplikasi e-commerce sederhana untuk menerapkan semua prinsip SOLID:

- User management dengan proper separation of concerns

- Order processing dengan extensible pricing strategies
- Payment system dengan multiple payment gateways
- Notification system dengan multiple channels
- Comprehensive unit tests dengan dependency mocking