Napredno PHP Programiranje

Dizajn obrasci i SOLID principi

Dizajn Obrasci

Što su dizajn obrasci?

Dizajn obrasci su standardizirana rješenja za česte probleme koje susrećemo u dizajnu softvera. Oni su poput predložaka koji se mogu prilagoditi za rješavanje određenih dizajnerskih problema u kodiranju.

Vrste dizajn obrazaca:

- **Creational**: Olakšavaju stvaranje objekata. Primjeri uključuju *Singleton*, *Factory*, *Builder*, *Prototype*.
- **Structural**: Olakšavaju dizajniranje arhitekture aplikacije. Primjeri uključuju *Adapter*, *Bridge*, *Decorator*, *Facade*.
- **Behavioral**: Usmjereni na komunikaciju između objekata. Primjeri uključuju *Observer*, *Strategy*, *Command*, *Iterator*.

Singleton obrazac

Singleton obrazac osigurava da klasa ima samo jednu instancu i pruža globalnu točku pristupa toj instanci.

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

    private function __construct() {
        // Inicijalizacija konekcije na bazu
    }

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

    public function query($sql) {
        // Izvršavanje SQL upita
    }
}

// Primjer korištenja
$db = Database::getInstance();
$db->query("SELECT * FROM users");
```

SOLID Principi

- **S: Single Responsibility Principle (SRP)**
  - Klasa bi trebala imati samo jedan razlog za promjenu, što znači da bi trebala imati samo jednu zadaću ili odgovornost.
- **O: Open/Closed Principle (OCP)**
  - Softver bi trebao biti otvoren za proširenje, ali zatvoren za izmjenu.
- **L: Liskov Substitution Principle (LSP)**
  - Objekti u programu bi trebali biti zamjenjivi s instancama njihovih podtipova bez utjecaja na točnost programa.
- **I: Interface Segregation Principle (ISP)**
  - Niti jedan klijent ne bi trebao biti prisiljen ovisiti o metodama koje ne koristi.
- **D: Dependency Inversion Principle (DIP)**
  - Visokorazinski moduli ne bi trebali ovisiti o niskorazinskim modulima. Oba bi trebala ovisiti o apstrakcijama.

Single Responsibility Principle

Princip jedinstvene odgovornosti navodi da bi svaka klasa trebala imati samo jednu odgovornost ili razlog za promjenu.
Ovo znači da bi klasa trebala biti fokusirana na jednu zadaću ili funkcionalnost.

User klasa samo upravlja korisničkim podacima, dok ***EmailService*** klasa upravlja slanjem e-pošte.

```php
// Loš pristup: Klasa "User" upravlja korisničkim podacima i logikom slanja e-pošte
class User {
    public $email;
    public $username;

    public function sendEmail($content) {
        // Logika za slanje e-pošte
    }
}

// Bolji pristup: Razdvajanje odgovornosti
class User {
    public $email;
    public $username;
}

class EmailService {
    public function sendEmail($user, $content) {
        // Logika za slanje e-pošte
    }
}
```

Open/Closed Principle

Princip otvorenosti/zatvorenosti kaže da bi software entiteti (klase, moduli, funkcije itd.) trebali biti otvoreni za proširenje, ali zatvoreni za izmjene.
To znači da bi trebali moći dodati nove funkcionalnosti bez mijenjanja postojećeg koda.

PaymentProcessor klasa je zatvorena za izmjene ali otvorena za proširenje kroz implementaciju ***PaymentGateway*** interfacea.

```php
interface PaymentGateway {
    public function processPayment($amount);
}

class PaypalPayment implements PaymentGateway {
    public function processPayment($amount) {
        // Proces plaćanja preko PayPal-a
    }
}

class CreditCardPayment implements PaymentGateway {
    public function processPayment($amount) {
        // Proces plaćanja kreditnom karticom
    }
}

class PaymentProcessor {
    private $paymentGateway;

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

    public function pay($amount) {
        $this->paymentGateway->processPayment($amount);
    }
}
```

Liskov Substitution Principle

Liskovljev princip supstitucije kaže da bi objekti programa trebali biti zamjenjivi s objektima njihovih podtipova bez utjecaja na točnost programa.
To znači da bi derivirani tipovi trebali moći zamijeniti svoje bazne tipove.

U ovom primjeru, korištenje ***Ostrich*** klase može izazvati probleme jer nojevi ne mogu letjeti. Ovaj dizajn krši LSP.

```php
class Bird {
    public function fly(){
        echo "Leti";
    }
}

class Duck extends Bird {}

class Ostrich extends Bird {
    public function fly() {
        throw new Exception("Ne može letjeti");
    }
}
```

Interface Segregation Principle

Princip segregacije sučelja sugerira da klijenti ne bi trebali biti prisiljeni ovisiti o sučeljima koja ne koriste.
To znači da bi sučelja trebala biti specifična, a ne generička.

Ovaj dizajn krši ISP jer ***RobotWorker*** ne treba *eat* metodu. Bolje bi bilo razdvojiti ova sučelja.

```php
interface Workable {
    public function work();
    public function eat();
}

class HumanWorker implements Workable {
    public function work() {
        // radi
    }
    public function eat() {
        // jede
    }
}

class RobotWorker implements Workable {
    public function work() {
        // radi
    }
    public function eat() {
        // Ova funkcija ne treba robotu
    }
}
```

Dependency Inversion Principle

Princip inverzije ovisnosti navodi da bi visoko razinski moduli ne trebali ovisiti o nisko razinski modulima. Oba bi trebala ovisiti o apstrakcijama.

FileManager ne ovisi direktno o ***DatabaseStorage***, već o ***Storage*** sučelju, što omogućava fleksibilnost i manju povezanost komponenti.

```php
interface Storage {
    public function save($data);
}

class DatabaseStorage implements Storage {
    public function save($data) {
        // Spremi podatke u bazu
    }
}

class FileManager {
    private $storage;

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

    public function saveFile($data) {
        $this->storage->save($data);
    }
}
```

Hvala na pažnji!