Charity Project

Software Design and Architecture Documentation

Project Author Name
`author@email.com`

August 30, 2025

Contents

1 Introduction

This document provides a detailed analysis of the software architecture and design patterns implemented in the Charity PHP project. The system is designed around a core architectural pattern (MVC) and several creational, structural, and behavioral design patterns that ensure separation of concerns, flexibility, and maintainability.

Each section that follows will describe a specific pattern, explain its role and implementation within the project, and present the relevant code excerpts directly from the source files for clear illustration.

2 Architectural Patterns

2.1 Model-View-Controller (MVC)

Intent: MVC is an architectural pattern that separates an application into three interconnected components: the Model, the View, and the Controller. This separation is crucial for organizing the codebase and decoupling the business logic from the user interface.

Implementation: The project's directory structure clearly reflects the MVC pattern:

- **Model:** Located in ‘src/models/‘, files like ‘users.php‘, ‘payments.php‘, and ‘events.php‘ manage the application's data, business rules, and logic. They are responsible for interacting with the database and representing the core entities of the system.
- **View:** Located in ‘src/views/‘, these are the PHP template files responsible for presenting data to the user. They contain the HTML structure and minimal PHP logic for displaying information provided by the controller. The ‘layout.php‘ file acts as a master template.
- **Controller:** Located in ‘src/controllers/‘, these files act as the intermediary between the Model and the View. They handle user input from the request, interact with the Model to fetch or update data, and then select the appropriate View to render the response.

The ‘Router‘ dispatches requests to the appropriate Controller, which then completes the cycle.

3 Creational Design Patterns

3.1 Singleton

Intent: The Singleton pattern ensures that a class has only one instance and provides a global point of access to it. This is used for managing shared resources like database connections, log files, and the application router.

Implementation: A generic ‘Singleton‘ abstract class provides the core functionality. The ‘Database‘, ‘Logger‘, and ‘Router‘ classes extend it. The static method ‘getInstance()‘ is the global access point to the single instance of each class.

```
% Filepath: src\core\singletons.php
<?php
abstract class Singleton
{
        private static $instances = [];

        protected function __construct() {}
        // ... (clone and wakeup are protected)

        public static function getInstance(): static
        {
                $cls = static::class;
                if (!isset(self::$instances[$cls])) {
                        self::$instances[$cls] = new static;
                }
                return self::$instances[$cls];
        }
}

class Database extends Singleton
{
        private $dbh;
    // ... constructor
        public static function getHandle(): PDO
        {
                return self::getInstance()->dbh;
        }
}
```

Listing 1: The Singleton base class and Database implementation.

4 Structural Design Patterns

4.1 Decorator

Intent: The Decorator pattern adds new responsibilities to an object dynamically without altering its class. It involves a "wrapper" object that contains the original object.

Implementation: ‘ProtectedHandler‘ is a Decorator that adds access control to any ‘Handler‘ object. It extends the abstract ‘RouteDecorator‘ and wraps a concrete ‘Handler‘. When a method (e.g., ‘GET‘, ‘POST‘) is called, ‘ProtectedHandler‘ first executes its security checks (is the user logged in? do they have the required role?). If the checks pass, it delegates the call to the wrapped ‘Handler‘ object, which proceeds with the core business logic.

```
% Filepath: src\core\ProtectedHandler.php
<?php
abstract class RouteDecorator extends Handler
{
        protected $wrappee;

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

class ProtectedHandler extends RouteDecorator
{
        private $requiredRole;

        public function __construct(Handler $wrappee, string $requiredRole)
        {
                parent::__construct($wrappee);
                $this->requiredRole = $requiredRole;
        }

        public function __call($method, $args)
        {
                if (!isset($_SESSION['user'])) {
                        http_error(401);
                }
                if (!$_SESSION['user'] instanceof $this->requiredRole) {
                        http_error(403);
```

```
            }
            return call_user_func_array([$this->wrappee, $method], $args);
        }
}
```

Listing 2: The Decorator for adding role-based access control.

```
% Filepath: src\controllers\admin\fundraisers.php
<?php
// ... (requires)
$handler = new class extends Handler
{
        public function GET() { /* ... */ }
        public function POST() { /* ... */ }
};

return new ProtectedHandler($handler, Admin::class);
```

Listing 3: Applying the decorator to a controller route.

5 Behavioral Design Patterns

5.1 Strategy

Intent: The Strategy pattern defines a family of interchangeable algorithms and encapsulates each one. This allows the algorithm to be selected at runtime.

Implementation: The application handles different payment methods using the Strategy pattern. The ‘PaymentMethod‘ interface defines the contract for all payment algorithms. Concrete classes like ‘CreditCardPayment‘ and ‘PayPalPayment‘ provide specific implementations. The ‘Donor‘ class (the "Context") holds a ‘PaymentMethod‘ object, allowing it to process payments without being coupled to a specific payment provider. The strategy is chosen in the ‘Donor::parse‘ method based on user data.

```
% Filepath: src\models\payments.php & src\models\users.php
<?php
// In payments.php
interface PaymentMethod
{
        public function pay(float $amount): void;
}

class CreditCardPayment implements PaymentMethod { /* ... */ }
class PayPalPayment implements PaymentMethod { /* ... */ }

// In users.php
class Donor extends User
{
        public PaymentMethod $paymentMethod;

        public function parse(array $data)
        {
                $this->paymentMethod = match ($data['payment_method'] ?? 'credit_card') {
                        'credit_card' => new CreditCardPayment,
                        'paypal' => new PayPalPayment,
                        'bank_transfer' => new BankTransferPayment,
                };
        }
    // ...
}
```

Listing 4: The Strategy interface and its context.

5.2 State

Intent: The State pattern allows an object to change its behavior when its internal state changes. The object appears to change its class.

Implementation: The ‘Donation‘ object transitions through a lifecycle (‘Created‘, ‘Pending‘, ‘Accepted‘, ‘Cancelled‘). The ‘DonationState‘ abstract class defines the common interface for all states. Each concrete state class (‘CreatedState‘, ‘PendingState‘, etc.) implements behavior specific to that state. The ‘Donation‘ object (the "Context") holds a reference to its current state object and delegates actions to it, transitioning to a new state when an action is completed.

```
% Filepath: src\models\payments.php
<?php
class Donation
{
        public DonationState $state;

        public function __construct(float $amount, Donor $donor)
        {
                // ...
                $this->state = new CreatedState($this);
        }

        public function proceed() { $this->state->proceed(); }
        public function cancel() { $this->state->cancel(); }
}

abstract class DonationState
{
        protected Donation $donation;
        public function __construct(Donation $donation) { /* ... */ }
        abstract public function proceed();
        abstract public function cancel();
}

class CreatedState extends DonationState
{
        public function proceed()
        {
                // ... logic for processing a new donation
                $this->donation->state = new PendingState($this->donation);
        }
    // ...
}
```

Listing 5: The State pattern for the Donation lifecycle.

5.3 Iterator

Intent: The Iterator pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Implementation: The project uses the Iterator pattern in two distinct ways:

1. **Controller Discovery:** In ‘index.php‘, ‘RecursiveDirectoryIterator‘ and ‘RecursiveIteratorIterator‘ are used to traverse the ‘controllers‘ directory to find and register all route handlers automatically.

2. **Data Fetching:** In ‘Donation::getAllPending()‘, the ‘yield‘ keyword is used to create a generator. A generator is PHP’s implementation of the Iterator pattern, allowing it to

fetch and return one pending donation at a time, which is highly memory-efficient for large datasets.

```php
% Filepath: src\index.php
<?php
$controllers_iterator = new RecursiveIteratorIterator(
        new RecursiveDirectoryIterator('controllers')
);

foreach ($controllers_iterator as $controller => $fileinfo) {
        if ($fileinfo->getExtension() === 'php') {
                // ... (logic to register route)
                $router->register($route, $controller);
        }
}
```

Listing 6: Using iterators for automatic route discovery.

```php
% Filepath: src\models\payments.php
<?php
class Donation {
    // ...
        public static function getAllPending()
        {
                // ... (database query)
                foreach ($rows as $row) {
                        // ... (object creation)
                        yield $donation;
                }
        }
    // ...
}
```

Listing 7: Using a generator (Iterator) for efficient data fetching.

5.4 Observer

Intent: The Observer pattern defines a one-to-many dependency between objects, so that when one object (the subject) changes state, all its dependents (observers) are notified and updated automatically.

Implementation: The ‘EventManager‘ (the "Subject") maintains a list of ‘EventListener‘ objects (the "Observers"). When a significant event occurs, such as a ‘Fundraiser‘ being saved, the ‘EventManager‘’s ‘notify()‘ method is called. This method iterates through all subscribed listeners and calls their ‘update()‘ method. This decouples the business logic of creating a fundraiser from subsequent actions like sending notifications.

```php
% Filepath: src\models\events.php
<?php
interface EventListener
{
        public function update(string $data);
        // ...
}

class EventManager extends Singleton
{
        private array $listeners;
        // ...
        public function notify(string $eventType, string $data)
        {
            if (!isset($this->listeners[$eventType])) return;
            foreach ($this->listeners[$eventType] as $listener) {
                $listener->update($data);
            }
        }
}

class Fundraiser
{
        public function save(): void
        {
            // ... (saves fundraiser to database)
            $eventManager = EventManager::getInstance();
            $eventManager->notify('fundraisers', "{$this->title} on {$this->date}");
        }
}
```

Listing 8: The Observer pattern implementation.

5.5 Template Method

Intent: The Template Method pattern defines the skeleton of an algorithm in a base class but lets subclasses override specific steps of the algorithm without changing its structure.

Implementation: The abstract ‘User‘ class defines the main algorithm for user creation in its constructor. The constructor is the "template method": it handles common initialization steps like setting the ID, email, etc., and then calls the abstract ‘parse()‘ method. Each concrete subclass (‘Admin‘, ‘Volunteer‘, ‘Donor‘, ‘Beneficiary‘) must provide its own implementation of ‘parse()‘ to handle the specific data structure associated with that user type.

```
% Filepath: src\models\users.php
<?php
abstract class User implements EventListener
{
        public function __construct(array $row)
        {
            $this->id = $row['id'];
            $this->name = $row['name'];
            $this->email = $row['email'];
            // ... common steps

            // Call the abstract method to be defined by subclasses
            $this->parse(json_decode($row['data'], true));
        }

        // This is the step that subclasses must implement
        abstract public function parse(array $data);
}

class Volunteer extends User
{
        public $skills, $availability;

        // Concrete implementation of the abstract step
        public function parse($data)
        {
            $this->skills = $data['skills'] ?? [];
            $this->availability = $data['availability'] ?? [];
        }
    // ...
}

```

```
class Donor extends User
{
        public PaymentMethod $paymentMethod;

        // A different implementation of the same abstract step
        public function parse(array $data)
        {
                $this->paymentMethod = match ($data['payment_method'] ?? 'credit_card') {
                        // ...
                };
        }
    // ...
}
```

Listing 9: The Template Method pattern in the User hierarchy.