ডিজাইন প্যাটার্ন

সফটওয়্যার ইঞ্জিনিয়ারিং এ ডিজাইন প্যাটার্ন হল কোডের সাধারণ সমস্যার রি-ইউজিবিলিটি বাড়ানোর জন্য এক প্রকারের নিয়মনীতি অথবা টেম্পলেট। যাতে করে সফটওয়্যার একটা নির্দিষ্ট আর্কিটেকচার এ তৈরি করা যায় আর কোডের পুনুরাব্রিত্তি ঠেকান যায়।

ডিজাইন প্যাটার্ন সাধারণত নিম্মলিখিত ক্যাটাগরীর হয়ে থাকেঃ

- Creational
- Structural
- Behavioural

সিঙ্গেলটোন/ Singleton

সিঙ্গেলটোন ডিজাইন প্যাটার্ন ক্রিয়েশনাল ডিজাইন প্যাটার্ন ক্যাটাগরির মধ্যে পরে। এই প্যাটার্নের মুল উদ্দেশ্য হল প্রতিটি ক্লাসের শুধু মাত্র একটিই ইন্সট্যান্স/অবজেক্ট থাকা।

ধরুন, Singleton নামে আমাদের একটা ফাইনাল ক্লাস আছে তাহলে সিঙ্গেলটোন প্যাটার্নে এই ক্লাসকে এমনভাবে ব্যবহার করতে হবে যেন নতুন কোন ইন্সট্যান্স/অবজেক্ট তৈরি না হয়ে একটিই থাকে আর ক্লাসটিকে ইনহেরিট ও করা না যায়, যা আমরা নিচের মত করে করতে পারিঃ

```
<?php

final class Singleton
{
   private static $instance;

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

```
        return self::$instance;
    }

    private function __construct()
    {

    }

    private function __clone()
    {

    }

    private function __wakeup()
    {

    }

    public function sayHi()
    {
        echo 'Hi';
    }
}

$singleton = Singleton::getInstance();

$singleton->sayHi();
```

এখানে ক্লাসটি বাইরে থেকে ইন্সট্যান্সিয়েট না করে getInstance() স্ট্যাটিক মেথডটি ডিক্লেয়ার করা হয়েছে যাতে ক্লাসের ইন্সট্যান্সটা রিটার্ন করে।

অর্থাৎ,

```
$singleton = new Singleton();
```

এর পরিবর্তে

```
$singleton = Singleton::getInstance();
```

ব্যবহার করা হয়েছে।

আর ক্লাসের ইন্সট্যান্স $instance নামে ভ্যারিয়েবল এ রাখা হয়েছে।

যেমনঃ

```
private static $instance;

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

   return self::$instance;
}
```

আবার ক্লাসের একাধিক ইন্সস্ট্যান্স তৈরিতে বাধা দিতে আমরা __clone() ও __wakeup() ম্যাজিক মেথডগুলি ব্যাবহার করেছি।

অবজার্ভার/Observer

অবজার্ভার ডিজাইন প্যাটার্ন বিহেভিওরাল টাইপের মধ্যে পরে। এটা pub/sub এর নিয়মে কাজ করে অর্থাৎ কোন অবজেক্ট কিংবা সাবজেক্ট এ পরিবর্তন হলে সেটা Publisher তৎক্ষন্যাত Subscriber দেরকে জানায় দিবে কিংবা নটিফাই করবে।

পিএইসপিতে অবজার্ভার প্যাটার্নটি প্রয়োগ করতে হলে যথাক্রমে SplSubject ও SplObserver ইন্টারফেইস ইমপ্লিমেন্ট করে সাবজেক্ট ও অবজার্ভার ২ টা ক্লাস লিখতে হয়। আর সাবস্ক্রাইব করা অবজার্ভারদেরকে স্টোর করে রাখার জন্য SplObjectStorage এই ক্লাসটিকে ব্যাবহার করা যেতে পারে।

উপরে উল্লেখিত SplSubject, SplObserver, SplObjectStorage হল পিএইসপির Standard PHP Library (SPL)

নিচে একটি Model নামক ক্লাস ও দুইটি অবজার্ভার ক্লাসের উদাহরণ দেয়া হলঃ

```
<?php

class Model implements SplSubject
{
  protected $observers;

  public function __construct()
  {
    $this->observers = new SplObjectStorage();
  }

  public function attach(SplObserver $observer)
  {
    $this->observers->attach($observer);
  }

  public function detach(SplObserver $observer)
  {
    $this->observers->detach($observer);
  }

  public function notify()
  {
    foreach ($this->observers as $observer) {
```

```
            $observer->update($this);
        }
    }

    public function __set($name, $value)
    {
        $this->data[$name] = $value;
        // notify the observers, that model has been updated
        $this->notify();
    }
}

class ModelObserver implements SplObserver
{
    public function update(SplSubject $subject)
    {
        echo get_class($subject) . ' has been updated' . '<br>';
    }
}

class Observer2 implements SplObserver
{
    public function update(SplSubject $subject)
    {
        echo get_class($subject) . ' has been updated' . '<br>';
    }
}

// Instantiate the model class for 2 different objects
$model1 = new Model();
$model2 = new Model();
```

```
// Instantiate the observers
$modelObserver = new ModelObserver();
$observer2 = new Observer2();

// Attach the observers to $model1
$model1->attach($modelObserver);
$model1->attach($observer2);

// Attach the observers to $model2
$model2->attach($observer2);

// Changing the subject properties
$model1->title = 'Hello World';
$model2->body = 'Lorem ipsum.....';
```

উপরে Model ক্লাসটি হল সাবজেক্ট ModelObserver ও Observer2 হল অবজার্ভার।

Model ক্লাসটি যেহেতু SplSubject ইন্টারফেইস ইমপ্লিমেন্ট করে লেখা হয়েছে কাজেই attach(), detach() ও notify() মেথডগুলা অবশ্যই থাকতে হবে।

অপরদিকে যেহেতু ModelObserver ও Observer2 ক্লাসগুলা SplObserver ইন্টারফেইস ইমপ্লিমেন্ট করে লেখা হয়েছে সেহেতু update() মেথডটি ক্লাসগুলাতে থাকতে হবে।

এবার আপনারা যদি SplSubject ও SplObserver ইন্টারফেইস ব্যাবহার না করে অবজার্ভার ডিজাইন প্যাটার্ন এর প্রয়োগ করতে চান সেটাও করতে পারবেন শুধুমাত্র আপনার বিষয় বস্তু ঠিক থাকলেই হল।

নিচে একটা উদাহরণ দেয়া হলঃ

```
<?php

class Model
{
```

```
    protected $observers;

    public function __construct()
    {
        $this->observers = new SplObjectStorage();
    }

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

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

    public function __set($name, $value)
    {
        $this->data[$name] = $value;
        // notify the observers, that model has been updated
        $this->notify();
    }
}

class Post extends Model
{
    public function insert($data)
```

```
    {
        // Store the data
        // Notify to observers
        $this->notify();
    }

    public function update($data)
    {
        // Update the model
        // Notify to observers
        $this->notify();
    }

    public function delete($id)
    {
        // Delete the model
        // Notify to observers
        $this->notify();
    }
}

class PostModelObserver
{
    public function update($subject)
    {
        echo get_class($subject) . ' has been updated' . '<br>';
    }
}

class Observer2
{
    public function update($subject)
```

```
  {
    echo get_class($subject) . ' has been updated' . '<br>';
  }
}

$post = new Post();

$post->setObservers([new PostModelObserver, new Observer2]);

$post->title = 'Hello World';
```

অ্যাডাপ্টার/ Adapter

সফটওয়ার ইঞ্জিনিয়ারিং এ আরেকটি বহুল প্রচলিত ডিজাইন প্যাটার্ন হল অ্যাডাপ্টার ডিজাইন প্যাটার্ন। এটি স্ট্রাকচারাল প্যাটার্নের মধ্যে পরে।

আমরা বাস্তব জীবনে সবাই অ্যাডাপ্টার শব্দটির সাথে পরিচিত। যেমনঃ মোবাইলের চার্জিং অ্যাডাপ্টার, কম্পিউটারের গ্রাফিক্স অ্যাডাপ্টার।

আর অ্যাডাপ্টার ডিজাইন প্যাটার্ন অনেকটা এই অ্যাডাপ্টারের ন্যায় কাজ করে অর্থাৎ আমরা যদি কম্পিউটারের গ্রাফিক্স কিংবা ভিজিএ অ্যাডাপ্টারের কথা চিন্তা করি তাহলে বলা যায় আমরা গেইম খেলার জন্য এক বিশেষ ধরনের অ্যাডাপ্টার ব্যবহার করি আবার সাধারণ কোন কাজের জন্য সাধারণ অ্যাডাপ্টার হলেই চলে কিন্তু বিষয়বস্তু দুইটারি সমান ভিডিও আউটপুট করা দুইটিই একটা কমন প্যাটার্নে তৈরি। আর মজার বিষয় হল এই অ্যাডাপ্টার গুলো আমাদের খুশি মত আমরা পরিবর্তন করতে পারি।

এবার ইমপ্লিমেন্টেশনের পরিভাষায়, ধরুন আমরা একটা পিএইচপি প্রজেক্ট কিংবা অ্যাপ্লিকেশন বানাবো যেখানে আমরা ডাটাবেস অ্যাডাপ্টার হিসেবে MySQL Adapter আর PDO Adapter ব্যাবহার করব যাতে করে ক্লাইন্ট সহজেই তার পছন্দের অ্যাডাপ্টারটি ব্যাবহার করতে পারে Database নামে আরেকটি অ্যাডাপ্টারের মাধ্যমে। এতে করে MySQL Adapter আর PDO Adapter গুলো খুব সহজেই পরিবর্তন করা যাবে।

নিচে একটা সম্পূর্ণ উদাহরণ দেয়া হলঃ

```
<?php
```

```
interface AdapterInterface
{
    public function query($sql);

    public function result();
}

class MySQLAdapter implements AdapterInterface
{
    protected $connection;

    protected $result;

    public function __construct($host, $username, $password, $dbname)
    {
        $this->connection = new mysqli($host, $username, $password, $dbname);
    }

    public function query($sql)
    {
        $this->result = $this->connection->query($sql);

        return $this;
    }

    public function result()
    {
        if (gettype($this->result) === 'boolean') {
            return $this->result;
        } elseif ($this->result->num_rows > 0) {
            $result = [];
```

```
        while ($row = $this->result->fetch_assoc()) {
            $result[] = $row;
        }

        return $result;
    } else {
        return [];
    }
  }
}

class PDOAdapter implements AdapterInterface
{
  protected $connection;

  protected $result;

  public function __construct($host, $username, $password, $dbname)
  {
    $this->connection = new PDO("mysql:host=$host;dbname=$dbname", $username, $password);
  }

  public function query($sql)
  {
    $query = $this->connection->prepare($sql);
    $exec = $query->execute();

    if ($query->columnCount() == 0) {
      $this->result = $exec;
    } else {
      $this->result = $query;
```

```
        }

        return $this;
    }

    public function result()
    {
        if (gettype($this->result) === 'boolean') {
            return $this->result;
        } else {
            $data = [];

            while ($row = $this->result->fetch(PDO::FETCH_ASSOC)) {
                $data[] = $row;
            }

            return $data;
        }
    }
}

class Database
{
    protected $adapter;

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

    public function query($sql)
    {
```

```
        return $this->adapter->query($sql);
    }

    public function result()
    {
        return $this->adapter->result();
    }
}

$mysql = new MySQLAdapter('localhost', 'root', '1234', 'demo');
$db = new Database($mysql);

$query = $db->query("SELECT * FROM users");
$result = $query->result();
var_dump($result);
```

এখানে AdapterInterface ইন্টারফেইস ব্যবহার করা হয়েছে যেটিকে ইমপ্লিমেন্ট করে যথাক্রমে MySQLAdapter ও PDOAdapter ডিক্লেয়ার করা হয়েছে যাতে দুইটারি ন্যাচার কিংবা কোডবেইস একই থাকে।

আবার ডাটাবেসকে অ্যাকসেস করার জন্য ও অ্যাডাপ্টারগুলাকে ব্যবহার করার জন্য Database নামে একটা ক্লাস ডিফাইন করা হয়েছে। আর এর ডিপেন্ডেন্সি ইনজেকশন হিসেবে AdapterInterface ব্যবহার করা হয়েছে যাতে করে কেবল মাত্র AdapterInterface ইমপ্লিমেন্ট করা ক্লাসের ইন্সটান্সই কন্সটারক্টরে পাস করা যায়।

এখানে আমরা MySQLAdapter কে ব্যবহার করেছি।

```
$mysql = new MySQLAdapter('localhost', 'root', '1234', 'demo');
$db = new Database($mysql);
```

আমরা চাইলে PDOAdapter ও ব্যবহার করতে পারি নিচের মত করে।

```
$pdo = new PDOAdapter('localhost', 'root', '1234', 'demo');
$db = new Database($pdo);
```

এতে করে অ্যাডাপ্টার গুলা Loosly Coupled/Highly Decoupled থাকে আর বর্তমানে এই টার্মটাকে খুবই প্রাধান্য দেয়া হয় বড় কোন অ্যাপ্লিকেশন কিংবা ফ্রেমওয়ার্ক তৈরি করতে গেলে।

ফ্যাক্টরী/Factory

ফ্যাক্টরী প্যাটার্ন এমন একটি প্যাটার্ন যা কম বেশি সব ধরনের অ্যাপ্লিকেশনে ব্যাবহৃত হয়ে থাকে এইটা ক্রিয়েশনাল প্যাটার্ন ক্যাটাগরীর মধ্যে পরে।

ফ্যাক্টরী প্যাটার্নের মূল উদ্দেশ্যই হল এর প্রোডাক্ট কিংবা চাইল্ড ক্লাসের অবজেক্ট তৈরি করে দেয়া। যেমন বাস্তব জীবনে যেভাবে ফ্যাক্টরীতে প্রোডাক্ট তৈরি হয়ে থাকে।

এই প্যাটার্ন ক্লাইন্টের কাছে অবজেক্ট ইনস্টানশিয়েট করার লজিক অদৃশ্যমান রাখে। আর অবজেক্টের ক্লাস গুলা একটা কমন ইন্টারফেইস কে ফলো করে বানানো থাকে।

এই প্যাটার্ন সাধারণত ৩ প্রকারেরঃ ১. সিম্পল ফ্যাক্টরী। ২. ফ্যাক্টরী মেথড। ৩. অ্যাবস্ট্রাক্ট ফ্যাক্টরী।

১. সিম্পল ফ্যাক্টরীঃ

ফ্যাক্টরী প্যাটার্নের মধ্যে সিম্পল ফ্যাক্টরী হচ্ছে সবচেয়ে সহজ প্যাটার্ন যদিও অফিশিয়ালি এই প্যাটার্ন ডিজাইন প্যাটার্ন হিসেবে স্বীকৃত না।

এই প্যাটার্নের নিয়ম মতে এর একটি ফ্যাক্টরী থাকবে আর একটি ফ্যাক্টরী একই সময় শুধুমাত্র একটাই প্রোডাক্ট তৈরি করবে অর্থাৎ একটিই ইনস্টান্স কিংবা অবজেক্ট রিটার্ন করবে।

নিচে একটা উদাহরণ দেয়া হলঃ

```
class CarFactory

{

   protected $brands = [];
```

```
    public function __construct()
    {
        $this->brands = [
            'mercedes' => 'MercedesCar',
            'toyota' => 'ToyotaCar',
        ];
    }

    public function make($brand)
    {
        if (!array_key_exists($brand, $this->brands)) {
            return new Exception('Not available this car');
        }

        $className = $this->brands[$brand];

        return new $className();
    }
}

interface CarInterface
{
    public function design();
    public function assemble();
    public function paint();
}

class MercedesCar implements CarInterface
{
    public function design()
    {
        return 'Designing Mercedes Car';
```

```
    }

    public function assemble()
    {
        return 'Assembling Mercedes Car';
    }

    public function paint()
    {
        return 'Painting Mercedes Car';
    }
}

class ToyotaCar implements CarInterface
{
    public function design()
    {
        return 'Designing Toyota Car';
    }

    public function assemble()
    {
        return 'Assembling Toyota Car';
    }

    public function paint()
    {
        return 'Painting Toyota Car';
    }
}

$carFactory = new CarFactory;
```

```
$mercedes = $carFactory->make('mercedes');
echo $mercedes->design() . '<br/>';
echo $mercedes->assemble() . '<br/>';
echo $mercedes->paint() . '<br/>';

echo '<br/>';

$toyota = $carFactory->make('toyota');
echo $toyota->design() . '<br/>';
echo $toyota->assemble() . '<br/>';
echo $toyota->paint() . '<br/>';
```

এখানে CarFactory নামে মূল ফ্যাক্টরী ক্লাস ডিফাইন করা হয়েছে যেটির মাধ্যমে একটা Car ইনস্টান্স তৈরি করা হবে। Car এর জন্য ২ টি ক্লাস যথাক্রমে MercedesCar ও ToyotaCar ডিফাইন করা হয়েছে যেগুলো CarInterface কে ফলো করেছে।

এবার চলুন CarFactory ক্লাসটিকে ইনস্টানশিয়েট করি।

```
$carFactory = new CarFactory;
```

এরপর ধরুন MercedesCar ক্লাসকে ফ্যাক্টরির মাধ্যমে ইনস্টানশিয়েট করব তাহলে টাইপ/প্যারামিটার হিসেবে mercedes দিতে হবে নিচের মত করে।

```
$mercedes = $carFactory->make('mercedes');
echo $mercedes->design() . '<br/>';
echo $mercedes->assemble() . '<br/>';
echo $mercedes->paint() . '<br/>';
```

অনুরূপ ভাবে ToyotaCar ক্লাসকে ইনস্টানশিয়েট করতে হলে

```
$toyota = $carFactory->make('toyota');
echo $toyota->design() . '<br/>';
```

```
echo $toyota->assemble() . '<br/>';
echo $toyota->paint() . '<br/>';
```

আর ডিফাইন না করা কোন ক্লাসের টাইপ দিলে সেটি এরর দেখাবে।

২. ফ্যাক্টরী মেথডঃ

ফ্যাক্টরী মেথড প্যাটার্ন অনেকখানি সিম্পল ফ্যাক্টরী প্যাটার্নের মতই শুধুমাত্র এর মূল পার্থক্য হল এটি তার সাব ক্লাস গুলোকে ক্লাস ইনস্টানশিয়েট করার স্বাধীনতা দিয়ে দেয়। আর এর একাধিক ফ্যাক্টরী থাকতে পারে।

নিচে একটা উদাহরণ দেয়া হলঃ

```
abstract class VehicleFactoryMethod
{
   abstract public function make($brand);
}

class CarFactory extends VehicleFactoryMethod
{
   public function make($brand)
   {
      $car = null;

      switch ($brand) {
         case "mercedes":
            $car = new MercedesCar;
            break;
         case "toyota":
            $car = new ToyotaCar;
            break;
      }
```

```
        return $car;
    }
}

class BikeFactory extends VehicleFactoryMethod
{
    public function make($brand)
    {
        $bike = null;

        switch ($brand) {
            case "yamaha":
                $bike = new YamahaBike;
                break;
            case "ducati":
                $bike = new DucatiBike;
                break;
        }

        return $bike;
    }
}

interface CarInterface
{
    public function design();

    public function assemble();

    public function paint();
}
```

```
interface BikeInterface
{
   public function design();

   public function assemble();

   public function paint();
}

class MercedesCar implements CarInterface
{
   public function design()
   {
      return 'Designing Mercedes Car';
   }

   public function assemble()
   {
      return 'Assembling Mercedes Car';
   }

   public function paint()
   {
      return 'Painting Mercedes Car';
   }
}

class ToyotaCar implements CarInterface
{
   public function design()
   {
      return 'Designing Toyota Car';
```

```
    }

    public function assemble()
    {
        return 'Assembling Toyota Car';
    }

    public function paint()
    {
        return 'Painting Toyota Car';
    }
}

class YamahaBike implements BikeInterface
{
    public function design()
    {
        return 'Designing Yamaha Bike';
    }

    public function assemble()
    {
        return 'Assembling Yamaha Bike';
    }

    public function paint()
    {
        return 'Painting Yamaha Bike';
    }
}

class DucatiBike implements BikeInterface
```

```
{
    public function design()
    {
        return 'Designing Ducati Bike';
    }

    public function assemble()
    {
        return 'Assembling Ducati Bike';
    }

    public function paint()
    {
        return 'Painting Ducati Bike';
    }
}

$carFactoryInstance = new CarFactory;

$mercedes = $carFactoryInstance->make('mercedes');
echo $mercedes->design() . '<br/>';
echo $mercedes->assemble() . '<br/>';
echo $mercedes->paint() . '<br/>';

echo '<br/>';

$toyota = $carFactoryInstance->make('toyota');
echo $toyota->design() . '<br/>';
echo $toyota->assemble() . '<br/>';
echo $toyota->paint() . '<br/>';

echo '<br/>';
```

```
$bikeFactoryInstance = new BikeFactory;


$yamaha = $bikeFactoryInstance->make('yamaha');
echo $yamaha->design() . '<br/>';
echo $yamaha->assemble() . '<br/>';
echo $yamaha->paint() . '<br/>';


echo '<br/>';


$ducati = $bikeFactoryInstance->make('ducati');
echo $ducati->design() . '<br/>';
echo $ducati->assemble() . '<br/>';
echo $ducati->paint() . '<br/>';
```

এখানে ফ্যাক্টরী মেথডের জন্য VehicleFactoryMethod নামে একটা অ্যাবস্ট্রাক্ট ক্লাস ডিফাইন করা হয়েছে যেটির সাব ক্লাস যথাক্রমে CarFactory ও BikeFactory আছে যেগুলা ভিন্ন ভিন্ন একক ফ্যাক্টরী। আবার প্রতিটি ফ্যাক্টরীর জন্য সিম্পল ফ্যাক্টরী প্যাটার্নের ন্যায় ইন্টারফেইস CarInterface ও BikeInterface ডিফাইন করা হয়েছে যেগুলোকে ইমপ্লিমেন্ট করে কংক্রিট ক্লাস অর্থাৎ ইনস্টানশিয়েট যোগ্য ক্লাস যথাক্রমে CarFactory এর আওতায় MercedesCar ও ToyotaCar এবং BikeFactory এর আওতায় YamahaBike ও DucatiBike ডিফাইন করা হয়েছে।

সুতরাং CarFactory ও BikeFactory ক্লাসগুলো নির্ধারণ করতে পারবে সে কোন ক্লাসকে ইনস্টানশিয়েট করবে।

নিচের কোডটি খেয়াল করলে বুঝতে পারবেন ২ টি আলাদা ফ্যাক্টরীর মাধ্যমে প্যারামিটার কিংবা Car এর ব্র্যান্ড পাস করে কাঙ্ক্ষিত অবজেক্ট কে পাওয়া যায়।$carFactoryInstance = new CarFactory;

```
$mercedes = $carFactoryInstance->make('mercedes');
echo $mercedes->design() . '<br/>';
echo $mercedes->assemble() . '<br/>';
echo $mercedes->paint() . '<br/>';


echo '<br/>';
```

```
$toyota = $carFactoryInstance->make('toyota');
echo $toyota->design() . '<br/>';
echo $toyota->assemble() . '<br/>';
echo $toyota->paint() . '<br/>';

echo '<br/>';

$bikeFactoryInstance = new BikeFactory;

$yamaha = $bikeFactoryInstance->make('yamaha');
echo $yamaha->design() . '<br/>';
echo $yamaha->assemble() . '<br/>';
echo $yamaha->paint() . '<br/>';

echo '<br/>';

$ducati = $bikeFactoryInstance->make('ducati');
echo $ducati->design() . '<br/>';
echo $ducati->assemble() . '<br/>';
echo $ducati->paint() . ‘<br/>';
```

৩. অ্যাবস্ট্রাক্ট ফ্যাক্টরীঃ

অ্যাবস্ট্রাক্ট ফ্যাক্টরী এমন একটি পদ্ধতি প্রদান করে যেখানে একটি মূল (অ্যাবস্ট্রাক্ট) ফ্যাক্টরী অনেকগুলো একক ফ্যাক্টরীকে একত্রিত করে রাখে।

এক কথায়, প্রথমে একটি অ্যাবস্ট্রাক্ট ফ্যাক্টরী অনেকগুলো প্রোডাক্ট ফ্যাক্টরী তৈরি করে এরপর প্রতিটি ফ্যাক্টরী একাধিক প্রোডাক্ট কিংবা অবজেক্ট তৈরি করে।

উল্লেখ্য, প্রতিটি প্রোডাক্ট ফ্যাক্টরী ক্লাসকে একটা কমন অ্যাবস্ট্রাক্ট ক্লাসকে এক্সটেন্ড করতে হবে অথবা একটা কমন ইন্টারফেইসকে ইমপ্লিমেন্ট করতে হবে। আবার প্রতিটি প্রোডাক্ট ফ্যাক্টরী ক্লাসে একাধিক আর একই মেথড থাকতে হবে।

নিচে একটা উদাহরণ দেয়া হলঃ

```
abstract class AbstractVehicleFactory
{
   abstract public function makeCar();
   abstract public function makeBike();
}

class BangladeshiFactory extends AbstractVehicleFactory
{
   public function makeCar()
   {
      return new ToyotaCar();
   }

   public function makeBike()
   {
      return new YamahaBike();
   }
}

class USAFactory extends AbstractVehicleFactory
{
   public function makeCar()
   {
      return new MercedesCar();
   }

   public function makeBike()
   {
      return new DucatiBike();
   }
```

```
}

abstract class AbstractVehicle
{
   abstract public function design();
   abstract public function assemble();
   abstract public function paint();
}

abstract class AbstractCarVehicle extends AbstractVehicle
{

}

abstract class AbstractBikeVehicle extends AbstractVehicle
{

}

class MercedesCar extends AbstractCarVehicle
{
   public function design()
   {
      return 'Designing Mercedes Car';
   }

   public function assemble()
   {
      return 'Assembling Mercedes Car';
   }

   public function paint()
```

```
    {
        return 'Painting Mercedes Car';
    }
}

class ToyotaCar extends AbstractCarVehicle
{
    public function design()
    {
        return 'Designing Toyota Car';
    }

    public function assemble()
    {
        return 'Assembling Toyota Car';
    }

    public function paint()
    {
        return 'Painting Toyota Car';
    }
}

class YamahaBike extends AbstractBikeVehicle
{
    public function design()
    {
        return 'Designing Yamaha Bike';
    }

    public function assemble()
    {
```

```
        return 'Assembling Yamaha Bike';
    }

    public function paint()
    {
        return 'Painting Yamaha Bike';
    }
}

class DucatiBike extends AbstractBikeVehicle
{
    public function design()
    {
        return 'Designing Ducati Bike';
    }

    public function assemble()
    {
        return 'Assembling Ducati Bike';
    }

    public function paint()
    {
        return 'Painting Ducati Bike';
    }
}

$bangladeshiFactoryInstance = new BangladeshiFactory;
$car = $bangladeshiFactoryInstance->makeCar();
echo $car->design() . '<br/>';
echo $car->assemble() . '<br/>';
echo $car->paint() . '<br/>';
```

```
echo '<br/>';

$bike = $bangladeshiFactoryInstance->makeBike();
echo $bike->design() . '<br/>';
echo $bike->assemble() . '<br/>';
echo $bike->paint() . '<br/>';

echo '<br/>';

$usaFactoryInstance = new USAFactory;
$car = $usaFactoryInstance->makeCar();
echo $car->design() . '<br/>';
echo $car->assemble() . '<br/>';
echo $car->paint() . '<br/>';

echo '<br/>';

$bike = $usaFactoryInstance->makeBike();
echo $bike->design() . '<br/>';
echo $bike->assemble() . '<br/>';
echo $bike->paint() . '<br/>';
```

উপরের কোডে AbstractVehicleFactory নামে একটা অ্যাবস্ট্রাক্ট ফ্যাক্টরী ক্লাস ডিফাইন করা হয়েছে যেখানে makeCar() ও makeBike() ২টা মেথড দেয়া আছে যাতে সাব ফ্যাক্টরী গুলো ওই মেথড গুলো ডিফাইন করে।

এখানে একটি মেথড Car অবজেক্ট তৈরি করতে আরেকটি Bike অবজেক্ট তৈরি করতে ব্যবহৃত হয়েছে যা সব গুলা ফ্যাক্টরীকেই করতে হবে। আর মূল বিষয় হল একেক ফ্যাক্টরী একেক ব্র্যান্ডের Car ও Bike অবজেক্ট তৈরি করবে।

যেমন এখানে আমরা BangladeshiFactory ফ্যাক্টরী ব্যাবহার করেছি যেটি ToyotaCar ও YamahaBike ক্লাসের অবজেক্ট তৈরি করবে। অনুরূপ ভাবে, USAFactory ফ্যাক্টরী MercedesCar ও DucatiBike ক্লাসের অবজেক্ট তৈরি করবে।

আবার আপনি চাইলে একটা ফ্যাক্টরীতে একাধিক ব্রান্ডের Car কিংবা Bike এর অবজেক্ট তৈরি করতে পারেন সেক্ষেত্রে রান্ডমলি কিংবা লজিক্যালি করতে হবে।

ডিপেন্ডেন্সি ইনজেকশন/ Dependency Injection (DI)

ডিপেন্ডেন্সি ইনজেকশন স্ট্রাকচারাল ডিজাইন প্যাটার্নের মধ্যে পরে। এইটা এমন একটা ডিজাইন প্যাটার্ন যা কোন ক্লাসের ডিপেন্ডেন্সি অর্থাৎ প্রয়োজনীয় অবজেক্ট গুলোকে রান টাইম কিংবা কম্পাইল টাইমে সহজে পরিবর্তনে সহায়তা করে।

এই প্যাটার্নের মূল উদ্দেশ্যই হল লুজলি কাপল আর্কিটেকচার ইমপ্লিমেন্ট করা যাতে করে একটা ভাল মানের অ্যাপ তৈরি করা যায়।

ডিপেন্ডেন্সি ইনজেকশন ডিজাইন প্যাটার্ন হল S.O.L.I.D Principle এর D যার পূর্ণ অর্থ Dependency Inversion Principle (DIP) যেটি Inversion of Control (IoC) কে অনুসরণ করে।

এখানে Dependency Inversion Principle বলতে Decoupling করাকে বুঝানো হয় আর Inversion of Control বলতে কিভাবে ডিপেন্ডেন্সি রিজল্ভ করা হবে সেটিকে বুঝায়। ডিপেন্ডেন্সি রিজল্ভ করতে Dependency Injection (DI) Container বা Inversion of Control (IoC) Container ব্যাবহৃত হয়ে থাকে।

আমরা সাধারণত একটি ক্লাসে অন্য ক্লাসের অবজেক্ট ব্যাবহার করলে নিচের মত করে হার্ডকোড করি যা হাইলি কাপল্ড থাকে।

```
class Database
{
   protected $adapter;

   public function __construct()
   {
      $this->adapter = new MySqlAdapter;
   }
}

class MysqlAdapter
```

```
{


}
```

আর ডিপেন্ডেন্সি ইনজেকশন ডিজাইন প্যাটার্নে কোন ক্লাস কিংবা ইন্টারফেইসকে টাইপ হিন্ট করে কনস্ট্রাক্টর কিংবা মেথডে ইঞ্জেক্ট করতে হয় নিচের মত করে।

```
class Database

{

   protected $adapter;


   public function __construct(MySqlAdapter $adapter)

   {

      $this->adapter = $adapter;

   }

}


class MysqlAdapter

{


}
```

উপরে MySqlAdapter ক্লাসকে ডিপেন্ডেন্সি হিসেবে রাখা হয়েছে। আর এই ডিপেন্ডেন্সি রিজল্ভ করতে হলে অবশ্যই MySqlAdapter এর অবজেক্ট কন্সট্রাক্টরের প্যারামিটারে দিতে হবে।

যেমনঃ

```
$mysqlAdapter = new MysqlAdapter;

$database = new Database($mysqlAdapter);
```

ডিপেন্ডেন্সি প্রধানত তিন ভাবে ইনজেক্ট করা যায়।

১. কন্সট্রাক্টর ইনজেকশনঃ

যা কনস্ট্রাক্টরের মাধ্যমে ইঞ্জেক্ট করা হয়।

```
public function __construct(MySqlAdapter $adapter)
{
   $this->adapter = $adapter;
}
```

২. সেটার ইনজেকশনঃ

যা কোন মেথডের প্যারামিটারে ইঞ্জেক্ট করা হয়।

```
public function setterMethod(MySqlAdapter $adapter)
{
   $this->adapter = $adapter;
}
```

৩. ইন্টারফেইস ইনজেকশনঃ

ইন্টারফেইসকে কোন কনস্ট্রাক্টরে অথবা সেটার মেথডে ইঞ্জেক্ট করা হয়।

```
public function __construct(AdapterInterface $adapter)
{
   $this->adapter = $adapter;
}
```

আমরা আমাদের প্রজেক্টে ডিপেন্ডেন্সি গুলোকে স্বয়ংক্রিয় ভাবে ইঞ্জেক্ট কিংবা রিজল্ভ করতে ডিপেন্ডেন্সি ইনজেকশন কন্টেইনার ব্যাবহার করব যা আগেই উল্লেখ করেছি। অনেক ফ্রেমওয়ার্কে এই কন্টেইনার সাধারণত বিল্ট-ইন দেয়া থাকে যেমনঃ Symfony, Laravel, Yii

আমরা সাধারণ প্রজেক্টের ক্ষেত্রে Pimple নামে কন্টেইনারটি ব্যাবহার করতে পারি। আবার আমি আমার কাজের জন্য খুব সহজ এবং অপ্টিমাইজ একটা কন্টেইনার বানিয়েছিলাম আপনারা চাইলে সেটি দেখতে পারেন থেকে। আশাকরি সোর্স কোড ও ডকুমেন্টেশন থেকে আপনারা ভাল ধারণা পাবেন।

ফ্যাসাড/ Facade

Facade ডিজাইন প্যাটার্ন স্ট্রাকচারাল ডিজাইন প্যাটার্নের মধ্যে পরে। ফ্যাসাড ডিজাইন প্যাটার্ন এর কাজ হল ক্লায়েন্ট এর কাছে একটা কমপ্লেক্স সিস্টেম বা ইন্টারফেইস হতে একটা সহজ ইন্টারফেইস প্রদান করা যাতে কমপ্লেক্স কিংবা আগলি কোড গুলো হিডেন অবস্থায় থাকে।

লেগাসি কিংবা কমপ্লেক্স কোন সিস্টেম এর কোডকে সহজ ভাবে উপস্থাপন করার দরকার হলে এই প্যাটার্ন ব্যাবহার করা হয়।

ধরুন আপনার সিস্টেম কিংবা অ্যাপ্লিকেশনে একটা কমপ্লেক্স লাইব্রেরি ব্যাবহার করার দরকার পরতেছে আর আপনি উক্ত কমপ্লেক্স পার্টকে সহজ করে তোলার জন্য একটা Wrapper বানিয়ে সেইটা করতে পারেন।

এবার নিচে একটা উদাহরণের মাধ্যমে প্যাটার্নটি বোঝানোর চেষ্টা করা হলঃ

```
<?php

class Cart
{
  public function addProducts($products)
  {
    // Product adding codes goes here
  }

  public function getProducts()
  {
    // Product retrieval codes goes here
  }
}
```

```
class Order
{
   public function process($products)
   {
      // Order processing codes goes here
   }
}

class Payment
{
   public function charge($charge)
   {
      // Additional charge codes goes here
   }

   public function makePayment()
   {
      // Payment method verify & payment codes goes here
   }
}

class Shipping
{
   public function calculateCharge()
   {
      // Calculation codes goes here
   }

   public function shipProducts()
   {
      // Ship process codes goes here
```

```
    }
}

class CustomerFacade
{
    public function __construct()
    {
        $this->cart = new Cart;
        $this->order = new Order;
        $this->payment = new Payment;
        $this->shipping = new Shipping;
    }

    public function addToCart($products)
    {
        $this->cart->addProducts($products);
    }

    public function checkout()
    {
        $products = $this->cart->getProducts();

        $this->totalAmount = $this->order->process($products);
    }

    public function makePayment()
    {
        $charge = $this->shipping->calculateCharge();
        $this->payment->charge($charge);

        $isCompleted = $this->payment->makePayment();
```

```
        if ($isCompleted) {
            $this->shipping->shipProducts();
        }
    }
}

$customer = new CustomerFacade;

$products = [
    [
        'name' => 'Polo T-Shirt',
        'price' => 40,
    ],
    [
        'name' => 'Smart Watch',
        'price' => 400,
    ],
];

$customer->addToCart($products);
$customer->checkout();
$customer->makePayment();
```

উপরের কোডটি খেয়াল করলে দেখতে পারবেন এখানে একটা ই-কমার্স অ্যাপ্লিকেশনের প্রসেস দেখানো হয়েছে। এর জন্য আমরা যথাক্রমে Cart, Order, Payment, Shipping ক্লাসগুলো ব্যাবহার করেছি আর ফ্যাসাড হিসেবে CustomerFacade ক্লাস ব্যাবহার করেছি। এখানে কোন প্রোডাক্টকে কার্টে যুক্ত করার জন্য Cart ক্লাসটি, অর্ডার প্রসেস করার জন্য Order ক্লাসটি, পেমেন্ট প্রসেস করার জন্য Payment ক্লাসটি আর প্রডাক্ট এর শিপিং হ্যান্ডল করার জন্য Shipping ক্লাসটি ব্যাবহার করেছি।

এখন মুল কথা হল আমরা যদি এসব কাজের জন্য প্রতিবার উক্ত ক্লাস গুলোকে বার বার কল করি তাহলে অনেক সময় সাপেক্ষ বেপার হয়ে পরবে আর স্ট্রাকচারটিও ভাল হবেনা। আর তাই এখানে ফ্যাসাড প্যাটার্নটি ব্যাবহার করা হয়েছে। যাতে ডেভেলপার কিংবা ক্লায়েন্ট হিসেবে সুধু মাত্র CustomerFacade ক্লাসটিকে ব্যাবহার করে উপরে উল্লেখিত সবগুলো কাজ অনায়াসে করা সম্ভব।

অতিরিক্ত বিষয় (লারাভেল ফ্যাসাড) ঃ

আমরা যারা লারাভেল ব্যাবহার করি তারা কম বেশি সবাই জানি লারাভেল এ অনেকগুলো বিল্ড-ইন ফ্যাসাড আছে কিংবা ব্যাবহার হয়। যেমনঃ DB, View, Event, Queue, Mail ইত্যাদি।

আমরা মূলত যেটা জানি তা হল লারাভেল এ ফ্যাসাড Statically কোন ক্লাসকে কল করার জন্য ব্যাবহার হয়। আসলে বিষয়টি ঠিক তেমন নয়। এখানে অনেক কমপ্লেক্স সিস্টেমকে হাইড করে আমাদের কাছে সহজ ভাবে উপস্থাপন করা হয়েছে তার সাথে লারাভেল ফ্যাসাড প্যাটার্নের সাথে __callStatic ম্যাজিক মেথডটি ব্যাবহার করা হয়েছে। যাতে ডেভেলপার কিংবা ক্লায়েন্টকে আলাদাভাবে ক্লাস ইন্সট্যানশিয়েট করতে না হয়।

নিচে একটা উদাহরণ দেয়া হলঃ

```
<?php

class Person
{
   public function getFullName()
   {
      return 'Sohel Amin';
   }
}

class PersonFacade
{
   static $instance;

   public static function __callStatic($method, $args)
   {
      if (null === static::$instance) {
         static::$instance = new Person;
      }

      $instance = static::$instance;
```

```
    switch (count($args)) {
      case 0:
        return $instance->$method();
      case 1:
        return $instance->$method($args[0]);
      case 2:
        return $instance->$method($args[0], $args[1]);
      case 3:
        return $instance->$method($args[0], $args[1], $args[2]);
      case 4:
        return $instance->$method($args[0], $args[1], $args[2], $args[3]);
      default:
        return call_user_func_array([$instance, $method], $args);
    }

  }
}

var_dump(PersonFacade::getFullName());
```

স্ট্রাটেজি/ Strategy

Strategy ডিজাইন প্যাটার্ন বিহেভিওরাল ডিজাইন প্যাটার্নের মধ্যে পরে। Strategy এর অর্থ হল কৌশল, কোন কিছু করতে গেলে তার জন্য কৌশল কিংবা এক গুচ্ছ পদক্ষেপ গ্রহণ করাই হল স্ট্রাটেজি।

প্রোগ্রামিং এর পরিভাষায়, একটি নির্দিষ্ট কাজ সম্পন্ন করতে ভিন্ন ভিন্ন অ্যালগরিদম নির্ধারণ করার স্বাধীনতা থাকাই স্ট্রাটেজি প্যাটার্ন। এই প্যটার্নকে আবার পলিসি প্যাটার্নও বলা হয়ে থাকে।

ধরুন, আপনি ঢাকা থেকে চট্টগ্রাম যাইতে চাচ্ছেন এরজন্য আপনি চাইলে বাস, ট্রেন কিংবা প্লেন এ করে যাইতে পারেন। এইক্ষেত্রে গন্তব্যস্থল একটিই কিন্তু এটা সম্পন্ন করতে ভিন্ন ভিন্ন স্ট্রাটেজি অনুসরণ করা যায়।

এবার চলুন একটা বাস্তব ভিত্তিক উদাহরণের মাধ্যমে প্যাটার্নটি বুঝা যাক। প্রথমে চলুন একটা ইন্টারফেইস বানিয়ে ভিন্ন ভিন্ন স্ট্রাটেজি ইমপ্লিমেন্ট করি নিচের মত করে।

```
interface TravelStrategy
{
   public function travel();
}

class BusTravelStrategy implements TravelStrategy
{
   public function travel()
   {
      // Bus travel strategy will goes here
   }
}

class TrainTravelStrategy implements TravelStrategy
{
   public function travel()
   {
      // Train travel strategy will goes here
   }
}

class PlaneTravelStrategy implements TravelStrategy
{
   public function travel()
   {
      // Plane travel strategy will goes here
```

```
    }
}
```

এবার মেইন কনটেক্সট ক্লাস হিসেবে Traveler নামক একটা ক্লাস ডিফাইন করি।

```
class Traveler
{
    protected $traveler;

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

    public function travel()
    {
        $this->traveler->travel();
    }
}
```

পরিশেষে, স্ট্রাটেজি পরিবর্তন করে সহজে আমরা আমাদের কার্য সম্পন্ন করতে পারি।

```
$traveler = new Traveler(new BusTravelStrategy());
$traveler->travel();

$traveler1 = new Traveler(new PlaneTravelStrategy());
$traveler1->travel();
```

ইটারেটর/ Iterator

ইটারেটর ডিজাইন প্যাটার্ন বিহেভিওরাল টাইপের মধ্যে পরে। এই প্যাটার্ন এর মুল উদ্দেশ্যই হচ্ছে ইটারেটরের ব্যাবহার করা। ইটারেটর একটা কন্টেইনার কিংবা অবজেক্ট এর ইলিমেন্টকে ট্রাভার্স করার জন্য সহায়তা করে আর এতে ভিতরের লজিক গুলো লুকানো অবস্থায় থাকে। যারফলে, আমরা কন্টেইনারে আমাদের পছন্দের মত ডাটা স্ট্রাকচার ব্যাবহার করতে পারি।

এবার চলুন আমরা কিভাবে এই প্যাটার্নটি ইমপ্লিমেন্ট করতে পারি। পিএইচপির একটা বিল্ড-ইন Iterator ইন্টারফেইস আছে আমরা সেটি ব্যাবহার করব।

সর্বপ্রথমে, আমরা ইলিমেন্ট বা আইটেম এর জন্য Book নামে একটা ক্লাস ডিফাইন করব।

```
class Book
{
   private $title;

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

   public function getTitle()
   {
      return $this->title;
   }
}
```

এবার কন্টেইনার এর জন্য BookList নামে একটা ক্লাস ডিফাইন করব।

```
class BookList implements Iterator, Countable
{
   private $books = [];

   private $currentIndex = 0;

   public function current()
```

```
{
    return $this->books[$this->currentIndex];
}

public function key()
{
    return $this->currentIndex;
}

public function next()
{
    $this->currentIndex++;
}

public function rewind()
{
    $this->currentIndex = 0;
}

public function valid()
{
    return isset($this->books[$this->currentIndex]);
}

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

public function addBook(Book $book)
{
    $this->books[] = $book;
```

```
    }

    public function removeBook(Book $bookToRemove)
    {
        foreach ($this->books as $key => $book) {
            if ($book->getTitle() === $bookToRemove->getTitle()) {
                unset($this->books[$key]);
            }
        }

        $this->books = array_values($this->books);
    }
}
```

এখানে Iterator ইন্টারফেসের জন্য যথাক্রমে current(), key(), next(), rewind() ও valid() মেথডগুলি ইমপ্লিমেন্ট করা হয়েছে আর Countable ইন্টারফেইসের এর জন্য count() মেথডটি ইমপ্লিমেন্ট করা হয়েছে যা ইলেমেন্ট কাউন্ট করতে সাহায্য করবে। আর ইলিমেন্ট অ্যাড আর রিমুভ করার জন্য addBook() ও removeBook() কাস্টম মেথডগুলি ব্যাবহার করা হয়েছে।

এবার কন্টেইনার ক্লাসটি ইন্সটানশিয়েট করে কিছু ইলিমেন্ট অ্যাড করে আমরা নিচের ন্যায় লুপের মাধ্যমে ইলিমেন্ট ট্রাভার্স করে অ্যাকসেস করতে পারি।

```
$bookList = new BookList();
$bookList->addBook(new Book('Design Pattern'));
$bookList->addBook(new Book('Head First Design Pattern'));
$bookList->addBook(new Book('Clean Code'));
$bookList->addBook(new Book('The Pragmatic Programmer'));

$bookList->removeBook(new Book('Design Pattern'));

foreach ($bookList as $book) {
    echo $book->getTitle() . PHP_EOL;
}
```

প্রক্সি/ Proxy

Proxy ডিজাইন প্যাটার্ন স্ট্রাকচারাল ডিজাইন প্যাটার্নের মধ্যে পরে। এই প্যাটার্ন শুরুর আগে আসুন আমরা “প্রক্সি” শব্দের অর্থ জেনে নেই। প্রক্সি এমন একটি প্রতিনিধি বা বস্তু যা অন্য বিষয় বস্তুর হয়ে কাজ করে।

অবজেক্ট অরিয়েন্টেড প্রোগ্রামিং এ প্রক্সি হলঃ একটি অবজেক্ট অন্য কোন অবজেক্টের হয়ে কাজ করা বা তাকে কন্ট্রোল করা।

প্রক্সি সাধারণত ৩ প্রকারেরঃ 1. Virtual Proxy: এই প্রক্সি মুল অবজেক্টকে ইন্সটানশিয়েট বা ইনিশিয়ালাইজ করতে বিলম্ব করে যতক্ষণ না দরকার পরে। 2. Remote Proxy: এই প্রক্সি কোন রিমুট লোকেশনে অবস্থিত কোন অবজেক্টকে রিপ্রেজেন্ট করে। যেমনঃ সার্ভার থেকে কোন অবজেক্টকে অ্যাকসেস করা। 3. Protection Proxy: এই প্রক্সি মুল অবজেক্টকে অ্যাকসেস করার আগে সেকুরিটি চেক করে। 4. Smart Proxy: এই প্রক্সি মুল অবজেক্টের রেফারেন্স নাম্বার ট্রাক করে এবং প্রয়োজন মত মেমোরি থেকে লোডিং অথবা ফ্রি করতে সহয়তা করে।

এখানে আমরা Virtual Proxy এর একটি উদাহরণ দেখব।

```
interface FileInterface
{
   public function content();
}

class RealFile implements FileInterface
{
   private $fileName;

   private $fileContent;

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

      $this->readFile();
```

```
    }

    private function readFile()
    {
        $this->fileContent = file_get_contents($this->fileName);
    }

    public function content()
    {
        return $this->fileContent;
    }
}

class ProxyFile implements FileInterface
{
    private $fileName;

    private $realFileObject;

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

    public function content()
    {
        // Lazy load the file using the RealFile class
        if (!$this->realFileObject) {
            $this->realFileObject = new RealFile($this->fileName);
        }

        return $this->realFileObject->content();
```

```
    }
}
```

উপরের কোডটি খেয়াল করলে আমরা দেখতে পাব একই ইন্টারফেইস FileInterface ব্যাবহার করে রিয়েল অবজেক্ট এর জন্য RealFile ও প্রক্সি অবজেক্টের জন্য ProxyFile নামক ক্লাস ইমপ্লিমেন্ট করা হয়েছে।

ProxyFile এর content() মেথডটি দেখলে বুঝতে পাব যে এর মাধ্যমে মুল RealFile ক্লাস এর ইন্সটানশিয়েট করা হয়েছে লেজিলোডিং পদ্ধতির মাধ্যমে যাতে অ্যাকসেস না করা পর্যন্ত ইন্সটানশিয়েট না করা হয়।

```
public function content()
{
    // Lazy load the file using the RealFile class
    if (!$this->realFileObject) {
        $this->realFileObject = new RealFile($this->fileName);
    }

    return $this->realFileObject->content();
}
```

এবার নিচের মত করে উভয় ক্লাসকে ইন্সটানশিয়েট করে কল করা হলে প্রথমে ভিন্ন ভিন্ন মেমোরি দখল করবে।

```
$realFile = new RealFile('/path/to/file.jpg');
var_dump(memory_get_usage()); // ~5Mb
$realFile->content();
var_dump(memory_get_usage()); // ~5Mb

$realFile->content();
var_dump(memory_get_usage()); // ~5Mb

$proxyFile = new ProxyFile('/path/to/file.jpg');
var_dump(memory_get_usage()); // ~350Kb
$proxyFile->content();
```

```
var_dump(memory_get_usage()); // ~5Mb


$proxyFile->content();

var_dump(memory_get_usage()); // ~5Mb
```

ডেকোরেটর/ Decorator

Decorator ডিজাইন প্যাটার্ন স্ট্রাকচারাল ডিজাইন প্যাটার্নের মধ্যে পরে। Decorator শব্দটি শুনলেই আমরা বুঝতে পারছি যে এটি কোন কিছুর প্রসাধক হিসেবে কাজ করে থাকে।

অবজেক্ট ওরিয়েন্টেডের ক্ষেত্রে ডেকোরেটর একটি নির্দিষ্ট অবজেক্টকে স্টাটিক্যালি অথবা ডায়নামিক্যালি সংযুক্তি বা পরিবর্তন করে থাকে।

এখন প্রশ্ন আসতে পারে আমরা কোন ক্লাসকে ইনহেরিট করেই তো এই কাজটি করতে পারি তাহলে কেন ডেকোরেটর ব্যাবহার করবো? ইনহেরিট্যান্স এর মাধ্যমে আমরা একটা ক্লাসকে পরিবর্তন করে থাকি তার মানে সাবক্লাস দিয়ে আমরা যতগুলো অবজেক্ট তৈরি করবো সবগুলাই সেইম হবে। অন্যদিকে ডেকোরেটর আমাদেরকে এই ক্ষেত্রে শুধুমাত্র কোন নির্দিষ্ট অবজেক্টে পরিবর্তন করতে ফ্লেক্সিবিলিটি দিয়ে থাকে।

এবার চলুন একটা উদাহরণ দেখা যাক।

```
interface EmailInterface
{
   public function body();
}

class Email implements EmailInterface
{
   public function body()
   {
      return 'Simple email body.';
   }
```

```
}

abstract class EmailDecorator implements EmailInterface
{
   public $email;

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

   abstract public function body();
}

class NewYearEmailDecorator extends EmailDecorator
{
   public function body()
   {
      return $this->email->body() . ' Additional text from deocorator.';
   }
}
```

উপরের কোডে খেয়াল করলে দেখতে পাবেন ইমেইল পাঠানোর জন্য একটি মুল ইন্টারফেইস EmailInterface আর এর কনক্রিট ক্লাস Email ইমপ্লিমেন্ট করা হয়েছে যা দিয়ে আমরা সিম্পল ইমেইল করতে পারি।

এবার ডেকোরেটর এর জন্য EmailDecorator অ্যাবস্ট্রাক্ট ক্লাস ডিফাইন করেছি আর এইটা কে ইনহেরিট করে NewYearEmailDecorator কনক্রিট ক্লাস ডিফাইন করেছি যার মাধ্যমে আমরা খুব সহজেই ইমেইলের অবজেক্টকে পরিবর্তন/সংযুক্তি করতে পারবো।

আমরা চাইলে EmailDecorator অ্যাবস্ট্রাক্ট ক্লাসটি ইনহেরিট করে আরও ক্লাস ডিফাইন করতে পারি।

এবার নিচের কোডটি দেখলে বুঝতে পারবেন কিভাবে অবজেক্টকে ডেকোরেট করা হয়েছে।

```
// Simple Email
$email = new Email();
var_dump($email->body());

// Decorated Email
$emailNewYearDecorator = new NewYearEmailDecorator($email);
var_dump($emailNewYearDecorator->body());
```
