**Major Unit: Design Patterns**

Introduction

Design patterns are solutions to recurring problems; guidelines on how to tackle certain problems. They are not classes, packages or libraries that you can plug into your application and wait for the magic to happen. These are, rather, guidelines on how to tackle certain problems in certain situations. Design patterns are solutions to recurring problems; guidelines on how to tackle certain problems. A general definition could be: In software engineering, a software design pattern is a general reusable solution to a commonly occurring problem within a given context in software design.

Types of Design Patterns

* Creational
* Structural
* Behavioral

**CREATIONAL DESIGN PATTERNS**

Creational patterns are focused towards how to instantiate an object or group of related objects. OR. In software engineering, creational design patterns are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation.

Some Creational design patterns are: Simple Factory, Factory Method, Abstract Factory, Builder, Prototype, Singleton.

**Simple Factory**

Consider, you are building a house and you need doors. It would be a mess if every time you need a door, you put on your carpenter clothes and start making a door in your house. Instead you get it made from a factory.

In plain words, Simple factory simply generates an instance for client without exposing any instantiation logic to the client.

Programmatic Example

First of all we have a door interface and the implementation

interface Door

{

public function getWidth(): float;

public function getHeight(): float;

}

class WoodenDoor implements Door

{

protected $width;

protected $height;

public function \_\_construct(float $width, float $height)

{

$this->width = $width;

$this->height = $height;

}

public function getWidth(): float

{

return $this->width;

}

public function getHeight(): float

{

return $this->height;

}

}

Then we have our door factory that makes the door and returns it

class DoorFactory

{

public static function makeDoor($width, $height): Door

{

return new WoodenDoor($width, $height);

}

}

And then it can be used as

$door = DoorFactory::makeDoor(100, 200);

echo 'Width: ' . $door->getWidth();

echo 'Height: ' . $door->getHeight();

When to Use?

When creating an object is not just a few assignments and involves some logic, it makes sense to put it in a dedicated factory instead of repeating the same code everywhere.

**Factory Method**

Defines an interface for creating an object, but let’s classes that > implement the interface decide which class to instantiate. The Factory > method lets a class defer instantiation to subclasses.

Programmatic Example

Taking our hiring manager example above. First of all we have an interviewer interface and some implementations for it.

interface Interviewer

{

public function askQuestions();

}

class Developer implements Interviewer

{

public function askQuestions()

{

echo 'Asking about design patterns!';

}

}

class CommunityExecutive implements Interviewer

{

public function askQuestions()

{

echo 'Asking about community building';

}

}

Now let us create our HiringManager

abstract class HiringManager

{

// Factory method

abstract protected function makeInterviewer(): Interviewer;

public function takeInterview()

{

$interviewer = $this->makeInterviewer();

$interviewer->askQuestions();

}

}

Now any child can extend it and provide the required interviewer

class DevelopmentManager extends HiringManager

{

protected function makeInterviewer(): Interviewer

{

return new Developer();

}

}

class MarketingManager extends HiringManager

{

protected function makeInterviewer(): Interviewer

{

return new CommunityExecutive();

}

}

and then it can be used as

$devManager = new DevelopmentManager();

$devManager->takeInterview(); // Output: Asking about design patterns

$marketingManager = new MarketingManager();

$marketingManager->takeInterview(); // Output: Asking about community building.

When to use?

Useful when there is some generic processing in a class but the required sub-class is dynamically decided at runtime.