PHP DESIGN PATTERNS

Practical Cheat Sheet

www.ApplicableProgramming.com

MVC

When: Separation of concerns in UI applications by splitting application in three main areas.
Usage: Web applications, GUI frameworks.
Key: Model <-> View <-> Controller

Model: Data & logic
View: UI representation
Controller: User input handling

SINGELTON

When: 1 and only 1 instance of a class.

Drawbacks: antipattern, breaks good practice, difficult to test, global state

Usage

- config
- database instance
- file instance
- Shared resources
- global state management

Reference code

```
Class Registry
{
    private static $instance = null;
    public static function instance(): self
    {
        self::$instance = is_null(self::$instance)
           ? new self() : self::$instance;
        return self::$instance;
    }
}
```

ADAPTER

When: Make different interfaces work together. Adapt one interface to already existing ecosystem.
Usage: Integrate legacy or external systems, refactor code.

```
interface Target {
   request();
}

class Adaptee {
   specificRequest();
}

class Adapter implements Target {
   /* adapt Adaptee to Target */
   request($adaptee){
      $adaptee->specificRequest();
   }
}
```

Drawbacks: Increased vode complexity

DEPENDENCY INJECTION:

When: a class needs to use another class - Reduce tight coupling between classes
Usage: controllers, services etc. Improve modularity, enable testing
Key: Inject dependencies via constructor or setter
Reference code:

```
class User($db){
    private $database;
    /* constructor injection */
    public function __construct($database){
      // set the $database
      $this->database = $database;
    }
    /* setter injection */
    public function setService($service) {
       // set and maybe use the $service
    }
}

// usage
$database = new Database();
$service = new Service();
$user = new User($database);
$user->setService($service);
```

ACTIVE RECORD - OBJECT RELATION MAPPER

Problem: Simplify CRUD operations with a database.
Usage: Object-relational mapping, data access.
Key: Model (represents table)

```
class Model {
 public function find();   // Retrieve
 public function insert(); // Create
 public function update(); // Update
 public function delete(); // Delete
}
```

Drawbacks: Limited flexibility (one-to-one mapping with database tables). Can lead to "fat models" (models with too much logic). Difficult with exceptions to the main functionality.

FACADE

Problem: Simplify complex operations with a unified interface. (usually operations that include multiple objects or classes)
Usage: Hide implementation details, simplifies usage.

```
class Facade {
  private $subsystemA;
  private $subsystemB;
  public function operation() {
    /* do something with subsystems */
  }
}
```

Drawbacks: May hide necessary details in some cases, additional layer

DECORATOR

Problem: Add or change functionality to objects dynamically during runtime, in different combinations
Usage: Unpredictable combination of changes (products, services doing calculations)
Key: Decorator <-> Component

```
interface Component { operation(); }
class Decorator implements Component {
  private $component;
  public __construct($component)
  public function operation() {
    /* change or add behavior */
  }
}
```

Drawbacks: Code complexity, difficult testing, Potential performance overhead (multiple layers of decoration).

SIMPLE FACTORY

Problem: Decouple object creation from its usage. Hide complex object creation or configuration from the object consumer.
Usage: Flexible and dynamic object instantiation.
Key: Creator <-> Product

```
class Creator {
   createComplexProduct();
}
class Product {
   /* product that is complex to create
*/
   oneOfManyConfigMethods();
}
```

Drawbacks: Code complexity: (additional classes or interfaces). Requires subclassing to create new products

OBSERVER

Problem: communication between many objects (that could also be missing from the system).
Usage: Event handling, plugins, dynamic codebase.
Key: Subject <-> Observer

```
interface Subject {
  attach(); detach(); notify();
}
interface Observer { update(); }
```

Drawbacks: Memory leaks (if observers aren't detached), performance issues (with many observers)

STRATEGY

Problem: changing different algorithms at runtime.
Usage: different services
Key: Context <-> Strategy

```
class Strategy { execute(); }

class Context {
  private $strategy;
  public function setStrategy($strategy){
    $this->strategy = $strategy;
  }
  public function doSomething(){
    $this->strategy->execute();
  }
}
```

Drawbacks: Increased number of objects (one per strategy).