**Scenario: Simple Content Management System (CMS)**

In your company, there's a growing need for a Content Management System (CMS) to manage articles, images, and videos for your website. The current implementation is quick and dirty, and it's becoming challenging to add new content types and features. New features would include things you would typically expect to find on a CMS and the corresponding webpage. You are tasked with using design patterns to improve the code.

**Step 1: Analysing the code (~20 minutes)**

Your first task is to critically analyse the provided ‘QuickAndDirtyCMS’ code.

* Your objective is to identify which Design Patterns may be applicable to improve the design.

*Note: Identify all design patterns you think are relevant, even if some overlap and it only makes sense to use one.*

**Step 2: Code Refinement (~20 minutes)**

Using the insights and observations from your analysis in Step 1:

* Propose an updated version of the code. Your coal is to select which relevant design patterns to use and update the code accordingly (feel free to use pseudo-code, but make sure it’s clear how the pattern is implemented as it is part of the learning opportunity to how patterns are applied in practice).

**Workshop Deliverables:**

At the end of this workshop, you should understand how some of the creational, structural, and behavioural design patterns are applied in practice, and you should be able to reason about the resulting improvements of using them.

*Note: Please remember that there's rarely a "one-size-fits-all" solution in software design. The goal of this exercise is to practice applying design patterns in real-world scenarios, not necessarily to reach a singular "correct" solution.*

**Creational Design Patterns** (object creation mechanisms, striving to create objects in a way suited to the situation)

* **The Singleton Pattern**: Ensures that a particular class has only one instance throughout the application and provides a global point of access to that instance. Useful for controlling access to shared resources.
* **The Factory Method Pattern**: A method for creating objects in a superclass but allows subclasses to alter the type of created objects. This provides a way to delegate the instantiation logic to child classes.
* **The Abstract Factory Method**: Produces families of related objects without specifying their concrete classes. It usually involves multiple Factory Methods, one for each type of object to be created.
* **The Builder Pattern**: Separates the construction of a complex object from its representation, allowing the same construction process to create different representations. Especially beneficial when an object needs to be created with many optional components or configurations.

**Structural Design Patterns** (provides ways to manage relationships and dependencies between classes and objects)

* **Adapter pattern:** Just like a physical adapter helps two different plugs connect, the Adapter pattern allows two incompatible interfaces to work together. It's all about creating an intermediary that transforms one interface into another.
* **Bridge pattern:** This pattern decouples an abstraction from its implementation, allowing both to vary independently. Think of it as a bridge between the functionality abstraction and its actual implementation.
* **Composite pattern:** This pattern lets you compose objects into tree structures to represent part-whole hierarchies. This allows clients to treat individual objects and compositions of objects uniformly.
* **Decorator pattern:** Enhancing an object's functionalities without altering its structure is where the Decorator pattern shines. It allows us to add responsibilities to objects dynamically.
* **Facade pattern:** Offering a simplified, unified interface to a set of interfaces in a subsystem, the Facade pattern helps hide the complexities of the system and provides a clearer, straightforward access point.

**Behavioural Design Patterns** (address responsibilities of objects and how they communicate)

* **Observer Pattern**: Enables an object (known as the subject) to maintain a list of its dependents (observers) and notify them of any state changes. It's widely used when a change in one object must be reflected in another without knowing how many objects need to be updated.
* **Chain of Responsibility Pattern**: Decouples the sender from the receiver by allowing more than one object to handle a request. The request is passed through a chain of potential handlers until either it's handled or reaches the end of the chain.
* **State Pattern**: Allows an object to change its behavior when its internal state changes, making it seem as if the object has changed its class.
* **Strategy Pattern**: Defines a family of algorithms, encapsulates each one, and makes them interchangeable. It lets the algorithm vary independently from clients that use it.
* **Command Pattern**: Encapsulates a request as an object, allowing users to parameterize objects with operations, queue requests, and support operations like undo.
* **Iterator Pattern**: Provides a way to access elements of an aggregate object sequentially without exposing its underlying representation.
* **Mediator Pattern**: Centralizes external communications between a set of objects to reduce direct dependencies and minimize the need for subclasses.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | 1 | *import* java.util.\*; | | 2 | public *class* QuickAndDirtyCMS { | | 3 | public static *void* main(*String*[] *args*) { | | 4 | // Content creation | | 5 | *Article* article = new *Article*("New Article", "This is a new article."); | | 6 | *Image* image = new *Image*("image1.jpg", "Image 1 description"); | | 7 | *Video* video = new *Video*("video1.mp4", "Video 1 description"); | | 9 | // Adding content to the website | | 10 | *Website* website = new *Website*(); | | 11 | website.addContent(article); | | 12 | website.addContent(image); | | 13 | website.addContent(video); | | 15 | // Displaying content | | 16 | *System*.out.println("Website Content:"); | | 17 | website.displayContent(); | | 18 | }   } | | 21 | *class* Article { | | 22 | private *String* title; | | 23 | private *String* body; | | 25 | public Article(*String* *title*, *String* *body*) { | | 26 | this.title = title; | | 27 | this.body = body; | | 28 | } | | 30 | public *void* display() { | | 31 | *System*.out.println("Article: " + title); | | 32 | *System*.out.println(body); | | 33 | }   } | | 36 | *class* Image { | | 37 | private *String* fileName; | | 38 | private *String* description; | | 40 | public Image(*String* *fileName*, *String* *description*) { | | 41 | this.fileName = fileName; | | 42 | this.description = description; | | 43 | } | | 45 | public *void* display() { | | 46 | *System*.out.println("Image: " + fileName); | | 47 | *System*.out.println("Description: " + description); | | 48 | }   } | | 51 | *class* Video { | | 52 | private *String* fileName; | | 53 | private *String* description; | | 55 | public Video(*String* *fileName*, *String* *description*) { | | 56 | this.fileName = fileName; | | 57 | this.description = description; | | 58 | } | | 60 | public *void* display() { | | 61 | *System*.out.println("Video: " + fileName); | | 62 | *System*.out.println("Description: " + description); | | 63 | }   } | | 66 | *class* Website { | | 67 | private *List*<*Object*> content = new *ArrayList*<>(); | | 69 | public *void* addContent(*Object* *item*) { | | 70 | content.add(item); | | 71 | } | | 73 | public *void* displayContent() { | | 74 | for (*Object* item : content) { | | 75 | if (item instanceof *Article*) { | | 76 | ((*Article*) item).display(); | | 77 | } else if (item instanceof *Image*) { | | 78 | ((*Image*) item).display(); | | 79 | } else if (item instanceof *Video*) { | | 80 | ((*Video*) item).display(); | | 81 | }   }   }   } | |

|  |  |
| --- | --- |
| **Creational Design Patterns**  **Singleton**  public class Singleton {  private static Singleton instance;  private Singleton() {}  public static Singleton getInstance() {  if (instance == null) {  instance = new Singleton();  }  return instance;  }  }  **Factory Method**  abstract class Product {}  class ConcreteProduct extends Product {}    abstract class Creator {  public abstract Product factoryMethod();  }  class ConcreteCreator extends Creator {  public Product factoryMethod() { return new ConcreteProduct(); }  }  **Abstract Factory**  interface Button {}  class WinButton implements Button {}  class MacButton implements Button {}    interface Factory {  Button createButton();  }  class WinFactory implements Factory {  public Button createButton() { return new WinButton(); }  }  class MacFactory implements Factory {  public Button createButton() { return new MacButton(); }  }  **Builder**  class Product {  String partA;  String partB;  // setters and getters...  }    class Builder {  Product product = new Product();  void buildPartA(String part) { product.setPartA(part); }  void buildPartB(String part) { product.setPartB(part); }  Product getResult() { return product; }  }  **Structural Design Patterns**  **Adapter Pattern**  **Object Adapter**  interface Target {  void request();  }  class Adaptee {  void specificRequest() { // Some specific logic here }  }  class Adapter implements Target {  Adaptee adaptee = new Adaptee();  public void request() { adaptee.specificRequest(); }  }  **Class Adapter**  interface Target {  void request();  }  class Adaptee {  void specificRequest() { // Some specific logic here }  }  class ClassAdapter extends Adaptee implements Target {  public void request() { specificRequest(); }  } | **Structural Design Patterns Continued**  **Bridge Pattern**  interface Implementation {  void operationImpl();  }  abstract class Abstraction {  Implementation impl;  abstract void operation();  }  class RefinedAbstraction extends Abstraction {  void operation() { impl.operationImpl(); }  }  **Composite Pattern**  import java.util.ArrayList;  import java.util.List;  interface Component {  void operation();  }  class Leaf implements Component {  private String name;  public Leaf(String name) {  this.name = name;  }  @Override  public void operation() {  System.out.println("Leaf: " + name);  } }  class Composite implements Component {  private List<Component> children = new ArrayList<>();  public void add(Component component) {  children.add(component);  }  @Override  public void operation() {  System.out.println("Composite:");  children.forEach(Component::operation);  } }  public class Client {  public static void main(String[] args) {  Component leaf1 = new Leaf("Leaf 1");  Component leaf2 = new Leaf("Leaf 2");  Composite composite = new Composite();  composite.add(leaf1);  composite.add(leaf2);  composite.operation();  } }  **Decorator Pattern**  interface Component {  void operation();  }  class ConcreteComponent implements Component {  public void operation() { // Original operation }  }  abstract class Decorator implements Component {  Component component;  }  class ConcreteDecorator extends Decorator {  public void operation() {  component.operation();  // Additional operation  }  }  **Facade Pattern**  class SubsystemA {  void operationA() { // Operation A logic }  }  class SubsystemB {  void operationB() { // Operation B logic }  }  class Facade {  SubsystemA a = new SubsystemA();  SubsystemB b = new SubsystemB();  void unifiedOperation() {  a.operationA();  b.operationB();  }  } |

|  |  |
| --- | --- |
| **Behavioural Patterns**  **Observer Pattern**  interface Observer {  void update();  }  class ConcreteObserver implements Observer {  public void update() {  // React to the update  }  }  class Subject {  List<Observer> observers = new ArrayList<>();  void addObserver(Observer o) {  observers.add(o);  }  void notifyAllObservers() {  for (Observer o : observers) {  o.update();  }  }  }  **Chain of Responsibility Pattern**  abstract class Handler {  Handler nextHandler;  void setNextHandler(Handler handler) {  this.nextHandler = handler;  }  abstract void handleRequest(String request);  }  class ConcreteHandlerA extends Handler {  void handleRequest(String request) {  // Handling logic or pass to next handler  }  }  **State Pattern**  interface State {  void handle();  }  class ConcreteStateA implements State {  public void handle() {  // Handle state A  }  }  class Context {  State currentState;  void setState(State state) {  this.currentState = state;  }  }  **Strategy Pattern**  interface Strategy {  void execute();  }  class ConcreteStrategyA implements Strategy {  public void execute() {  // Execute strategy A  }  }  class Context {  Strategy strategy;  void setStrategy(Strategy strategy) {  this.strategy = strategy;  }  void executeStrategy() {  strategy.execute();  }  } | **Behavioural Patterns continued**  **Command Pattern**  interface Command {  void execute();  }  class ConcreteCommandA implements Command {  public void execute() {  // Execute action A  }  }  class Invoker {  Command command;  void setCommand(Command cmd) {  this.command = cmd;  }  void invoke() {  command.execute();  }  }  **Iterator Pattern**  interface Iterator {  boolean hasNext();  Object next();  }  class ConcreteIterator implements Iterator {  public boolean hasNext() {  // Check if there's a next element  }  public Object next() {  // Return the next element  }  }  **Mediator Pattern**  interface Mediator {  void notify(String event, Component component);  }  class ApplicationMediator implements Mediator {  public void notify(String event, Component component) {  // Handle component notifications  }  }  abstract class Component {  Mediator mediator;  public Component(Mediator mediator) {  this.mediator = mediator;  }  public abstract void doAction();  }  class Button extends Component {  public Button(Mediator mediator) {  super(mediator);  }  public void doAction() {  // Do button related action  mediator.notify("ButtonPressed", this);  }  }  class Textbox extends Component {  public Textbox(Mediator mediator) {  super(mediator);  }  public void doAction() {  // Do textbox related action  mediator.notify("TextboxUsed", this);  }  } |