Here are the detailed answers based on logical reasoning and references to software engineering principles from Ian Sommerville and Roger Pressman.

**(a) Design Pattern, Coupling, and Cohesion**

**Design Pattern:**  
A design pattern is a reusable solution to a commonly occurring problem in software design. It provides a proven approach to solving specific design issues while improving code maintainability and flexibility. Examples include Singleton, Factory, and Observer patterns.

**Coupling:**  
Coupling refers to the level of dependency between different modules or components in a system.

* **High coupling** means that modules are highly dependent on each other, making changes difficult.
* **Low coupling** means that modules are independent, improving flexibility and maintainability.

**Cohesion:**  
Cohesion refers to how closely related the functionalities within a module are.

* **High cohesion** means that a module performs a single, well-defined task, improving readability and maintainability.
* **Low cohesion** means that a module performs unrelated tasks, making it harder to understand and modify.

**(b) UML and Implementation for User Interface Toolkit (Borders & Scrollbars)**

This scenario can be implemented using the **Decorator Pattern**, which allows adding features (like borders and scrollbars) dynamically to UI components.

**UML Diagram for Decorator Pattern**

+-------------------+

| Window |

+-------------------+

| display() |

+-------------------+

▲

│

+-------------------+

| WindowDecorator |

+-------------------+

| display() |

+-------------------+

▲

│

+-------------------+ +-----------------+

| ScrollBar |------->| BorderDecorator |

+-------------------+ +-----------------+

| display() | | display() |

+-------------------+ +-----------------+

**Java Implementation**

// Base Component

interface Window {

void display();

}

// Concrete Component

class SimpleWindow implements Window {

public void display() {

System.out.println("Displaying a simple window.");

}

}

// Decorator

abstract class WindowDecorator implements Window {

protected Window window;

public WindowDecorator(Window window) {

this.window = window;

}

public void display() {

window.display();

}

}

// Concrete Decorators

class ScrollBar extends WindowDecorator {

public ScrollBar(Window window) {

super(window);

}

public void display() {

super.display();

System.out.println("Adding scrollbar.");

}

}

class BorderDecorator extends WindowDecorator {

public BorderDecorator(Window window) {

super(window);

}

public void display() {

super.display();

System.out.println("Adding border.");

}

}

// Usage

public class DecoratorExample {

public static void main(String[] args) {

Window window = new SimpleWindow();

Window borderedWindow = new BorderDecorator(window);

Window scrollWindow = new ScrollBar(borderedWindow);

scrollWindow.display(); // Output: Displaying a simple window. Adding border. Adding scrollbar.

}

}

**(c) Chain of Responsibility Pattern**

This pattern allows multiple objects to process a request. The request is passed along a chain until one object handles it.

**UML Diagram**

+----------------+

| Handler |

+----------------+

| handleRequest()|

+----------------+

▲

│

+----------------+

| ConcreteHandler|

+----------------+

| handleRequest()|

+----------------+

**Java Implementation**

// Handler Interface

abstract class Handler {

protected Handler next;

public void setNext(Handler next) {

this.next = next;

}

public abstract void handleRequest(int level);

}

// Concrete Handlers

class ConcreteHandlerA extends Handler {

public void handleRequest(int level) {

if (level < 5)

System.out.println("Handler A handled request.");

else if (next != null)

next.handleRequest(level);

}

}

class ConcreteHandlerB extends Handler {

public void handleRequest(int level) {

System.out.println("Handler B handled request.");

}

}

// Usage

public class ChainOfResponsibilityExample {

public static void main(String[] args) {

Handler handlerA = new ConcreteHandlerA();

Handler handlerB = new ConcreteHandlerB();

handlerA.setNext(handlerB);

handlerA.handleRequest(4); // Handler A handles it

handlerA.handleRequest(10); // Handler B handles it

}

}

**(d) Singleton Design Pattern**

**Why Singleton?**

* Ensures a single instance of a class.
* Controls access to shared resources like databases or logging.
* Reduces memory usage.

**Java Implementation**

class Singleton {

private static Singleton instance;

private Singleton() {} // Private constructor

public static Singleton getInstance() {

if (instance == null) {

instance = new Singleton();

}

return instance;

}

public void showMessage() {

System.out.println("Singleton instance!");

}

}

// Usage

public class SingletonExample {

public static void main(String[] args) {

Singleton s = Singleton.getInstance();

s.showMessage();

}

}

**(e) Code Smells and Strategy Pattern for Pricing System**

**Code Smells:**  
Code smells are signs of poor design that make the code difficult to maintain. Examples:

* **Duplicate Code** (same logic repeated in multiple places)
* **Long Methods** (methods doing too many tasks)
* **Large Classes** (too many responsibilities)

**UML for Strategy Pattern**

+----------------------+

| PricingStrategy |

+----------------------+

| calculatePrice() |

+----------------------+

▲

│

+----------------------+

| DiscountStrategy |

+----------------------+

| calculatePrice() |

+----------------------+

**Java Implementation**

// Strategy Interface

interface PricingStrategy {

double calculatePrice(double basePrice);

}

// Concrete Strategies

class NoDiscount implements PricingStrategy {

public double calculatePrice(double basePrice) {

return basePrice;

}

}

class SeasonalDiscount implements PricingStrategy {

public double calculatePrice(double basePrice) {

return basePrice \* 0.8;

}

}

// Context Class

class Product {

private PricingStrategy strategy;

public Product(PricingStrategy strategy) {

this.strategy = strategy;

}

public double getPrice(double basePrice) {

return strategy.calculatePrice(basePrice);

}

}

// Usage

public class StrategyExample {

public static void main(String[] args) {

PricingStrategy discount = new SeasonalDiscount();

Product product = new Product(discount);

System.out.println("Final Price: " + product.getPrice(100));

}

}

**(f) Factory Pattern for Virtual Pet Game**

To create families of related objects (pets and habitats), the **Factory Pattern** is used.

**Java Implementation**

// Product Interface

interface Pet {

void live();

}

// Concrete Products

class JunglePet implements Pet {

public void live() {

System.out.println("Jungle Pet lives in a jungle!");

}

}

class TundraPet implements Pet {

public void live() {

System.out.println("Tundra Pet survives in the cold!");

}

}

// Factory Class

class PetFactory {

public static Pet createPet(String type) {

if (type.equalsIgnoreCase("jungle")) {

return new JunglePet();

} else if (type.equalsIgnoreCase("tundra")) {

return new TundraPet();

}

return null;

}

}

// Usage

public class FactoryExample {

public static void main(String[] args) {

Pet pet = PetFactory.createPet("jungle");

pet.live();

}

}