Object Oriented Modelling and Design

BCS1430

Dr. Ashish Sai

Week 2 Lecture 1 & 2

BCS1430.ashish.nl

EPD150 MSM Conference Hall

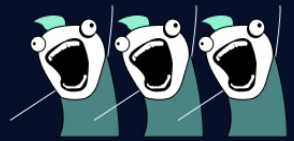
Table of contents

Lecture 2

> Learning how to design OO software

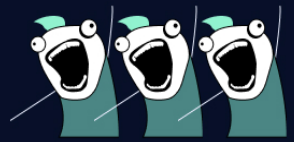
WHO WE ARE?

CLIENTS!



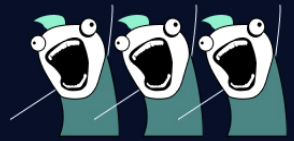
WHAT DO WE WANT?

WE DON'T KNOW!

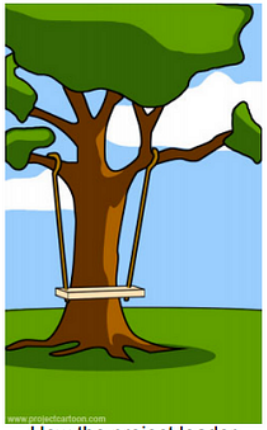


WHEN DO WE WANT IT?

NOW!



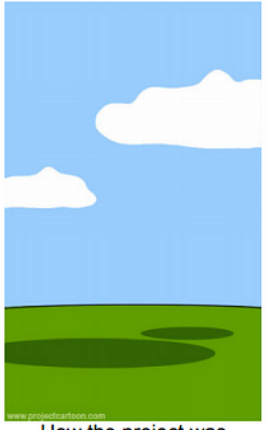

How the customer explained it

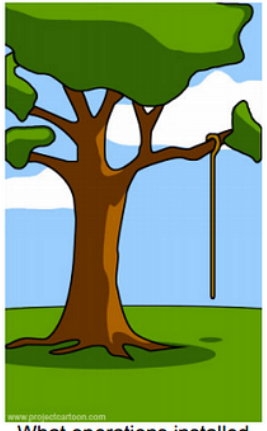

How the project leader understood it

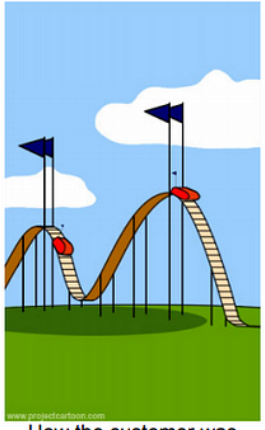

How the analyst designed it

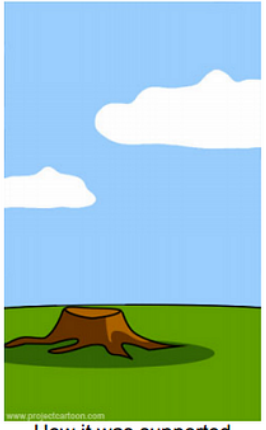

How the programmer wrote it


How the business consultant described it


How the project was documented


What operations installed


How the customer was billed


How it was supported


What the customer really needed

Object Interaction Analysis

- Object Interaction Analysis is a technique used to understand how objects in the system will communicate and collaborate.
- Interaction diagrams are used to visualize these interactions [1].

1. More on these diagrams in the next lecture.

Object Interaction Analysis

- For example, in an airline reservation system, understanding how a ‘Passenger’ object interacts with ‘Flights’, ‘Tickets’, and ‘Payments’ objects is crucial for designing a functional system.

Assigning Responsibilities with CRC Cards

Class-Responsibility-Collaborator (CRC) Cards: CRC cards is a simple tool used to define the behaviors and interactions of classes.

Components of a CRC Card:

- **Class Name:** The entity or concept being modeled.
- **Responsibilities:** What the class should know or do (its behavior).
- **Collaborators:** Other classes this class interacts with or uses.

CRC Card Example - ShoppingCart

Class Name: ShoppingCart

Responsibilities:

- Add Item: Include a new product in the cart.
- Calculate Total: Compute the total cost of items in the cart.
- Remove Item: Take out a product from the cart.
- Checkout: Initiate the purchasing process.

Collaborators:

- Product: Items that can be added to the shopping cart.
- User: The customer who owns the shopping cart.

CRC Card Example - ShoppingCart

```java
class ShoppingCart {
    private List<Product> products;
    private User owner;

    void addItem(Product product) { /*...*/ }
    double calculateTotal() { /*...*/ }
    void removeItem(Product product) { /*...*/ }
    void checkout() { /*...*/ }
}

class Product { /* Product details */ }
class User { /* User details */ }
```

The ShoppingCart class has clear responsibilities like managing items and calculating totals, and it collaborates with Product and User classes to fulfill these responsibilities.

Design

Good Design in Software

- Beyond aesthetics, good software design is about crafting solutions that are effective, efficient, and maintainable.
- **Simplicity, Consistency, Modularity, Usability, Maintainability, Robustness**

Simplicity

- Simplicity means focusing on essential elements, making software more understandable and less error-prone.

In Practice:

- A Java method that performs a single, well-defined function is a good example. Easy to understand, test, and maintain.

Simplicity

```java
// Method to add two numbers

public class Addition {

    // Entry method to demonstrate addition complexity
    public static void main(String[] args) {
        int number1 = 5;
        int number2 = 10;

        int result = Add(number1, number2);
        System.out.println("The result is: " + result);
    }

    // addition method
    public static int Add(int a, int b) {
        // Initialize sum
        int sum = 0;

        // Convert integers to strings
        String strA = Integer.toString(a);
```

Simplicity

```
1 // Simple method to add two numbers
2 int add(int a, int b) {
3     return a + b;
4 }
```

- Reduces complexity, enhances maintainability, and improves user experience.

Consistency

- Consistency involves uniformity in visual elements, terminology, and behavior.

In Practice:

- A Java project where all variables, methods, and classes strictly adhere to `CamelCase` naming conventions.

(In)Consistency

```
// The Variable Naming Convention Extravaganza

int numberOfCatsOwnedByAunt = 5;
// Clearly, someone loves their aunt... and cats

double BitcoinInvestmentValue2024 = 42069.42;
// PascalCase:To the moon, they said

boolean is_this_variable_named_correctly = false;
// snake_case: A philosophical inquiry

String favorite-ice-cream-flavor = "MintChocolateChip";
// kebab-case: A contentious choice (Syntax Error!)

// And in a bold move to challenge the very fabric of naming conventions...

String uniFied_NamingConvention2024 = "AbsolutelyNotRecommended";
// An optimist's futile attempt

```

Consistency

```

// The Variable Naming Convention Harmonization

int numberOfCatsOwnedByAunt = 5;
// Consistent CamelCase: Reflecting a fondness for one's aunt... and cats

double bitcoinInvestmentValue2024 = 42069.42;
// CamelCase: Optimistically aiming for the moon

boolean isThisVariableNamedCorrectly = false;
// CamelCase: Pondering the philosophical depths of naming correctness

String favoriteIceCreamFlavor = "MintChocolateChip";
// CamelCase: A choice that's as bold as it is divisive

// Unifying the naming convention to restore order from chaos
String unifiedNamingConvention2024 = "HighlyRecommended";
// CamelCase: A testament to the power of consistency


```

Modularity

- Modularity means designing systems as separate, interchangeable components.

In Practice:

- Java packages organizing classes by functionality, allowing independent development and testing.

Modularity

```java
// Package for animal management
package com.zoo.animalcare;
public class AnimalFeeder {
    // Feed the animals, or they start considering you as the next meal
}

// Package for zoo staff management
package com.zoo.staffmanagement;
public class StaffScheduler {
    // Schedule staff or face the chaos of a zoo without keepers
}

// Separate package for gift shop inventory
package com.zoo.giftshop;
public class InventoryManager {
    // Keep the plushies stocked, or brace for a toddler tantrum apocalypse
}

// And for the adventurous souls...
package com.zoo.emergencyprotocols;
```

Impact: - Enhances flexibility, manageability, and scalability.

Usability

- Usability focuses on making software intuitive and accessible based on users' needs and limitations.

In Practice:

- A Java web application with a straightforward, well-organized layout, clear instructions, and responsive feedback.

Usability

- Imagine a Java application designed for a very niche market: professional couch potatoes. The app, named “LazyLogger,” helps users track their daily inactivity periods, favorite couch spots, and snack consumption with minimal effort required from the user.

Usability

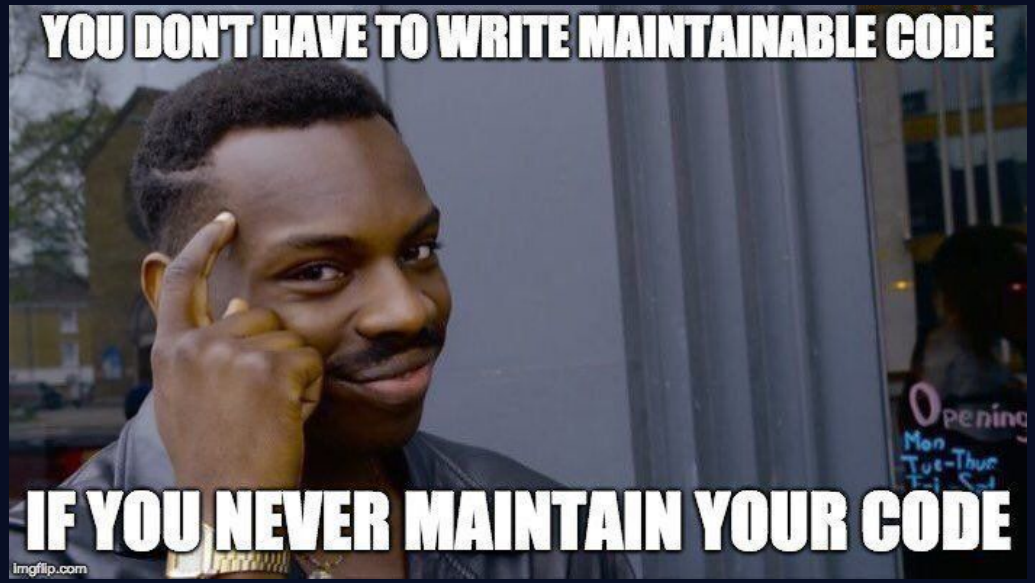
```java
// Ultra-simple UI for the ultimate couch potato experience
public class LazyLoggerUI {

    public void logInactivity() {
        if (isUserAwake()) {
            displayMessage("Welcome! Just press any key to log today's laziness.");
            waitForAnyKeyPress();
            logLaziness("Another day successfully wasted.");
        } else {
            // Auto-log for those too lazy to even press a key
            logLaziness("User too lazy to interact. Laziness logged automatically.");
        }
    }

    private boolean isUserAwake() {
        // Implement complex algorithm to detect user's consciousness
        return Math.random() > 0.5; // 50/50 chance
    }

    private void waitForAnyKeyPress() {
```

- Makes software easy and pleasant to use, enhancing user satisfaction.


YOU DON'T HAVE TO WRITE MAINTAINABLE CODE
IF YOU NEVER MAINTAIN YOUR CODE
imgflip.com

Maintainability

Concept:

- Maintainable design allows for easy understanding, correction, adaptation, and enhancement.

In Practice:

- Writing clean, well-documented Java code and adhering to architectural patterns.

Maintainability

```java
// Movie recommendation for the world's most indecisive person

/**
 * Suggests a movie based on the user's mood and weather.
 * This method embodies the art of overthinking movie night.
 * @param mood The user's current mood, e.g., "happy", "sad".
 * @param weather The current weather, e.g., "rainy", "sunny".
 * @return A string suggesting a movie, because choosing is hard.
 */
public String suggestMovie(String mood, String weather) {
    // If it's raining and the user is sad, recommend a comedy
    if ("rainy".equals(weather) && "sad".equals(mood)) {
        return "Watching 'Monty Python and the Holy Grail' will lift your spirits!";
    }
    // If it's sunny and the user is happy, recommend an adventure movie
    else if ("sunny".equals(weather) && "happy".equals(mood)) {
        return "It's a perfect day for 'Indiana Jones'!";
    }
    // For all other cases, recommend something random
    // because who doesn't love a surprise movie pick?
```

Impact:

- Ensures software can evolve and adapt

Robustness

Concept:

- Robustness means the system’s ability to handle errors and unexpected situations gracefully.

In Practice:

- Java applications implementing error handling and validation to manage unexpected inputs or disruptions.

Robustness

```java
// "FutureSeer", attempts to predict daily events with a mix of technology and,
// let's say, less scientific methods.

/**
 * Attempts to predict the user's future by combining high-tech algorithms
 * with a touch of mystical randomness.
 */
public String predictFuture() {
    try {
        // Pretend to perform some complex calculation involving astrology, machine learning,
        if (Math.random() > 0.5) {
            return "Good fortune awaits you today!";
        } else {
            throw new UncertainFutureException("The future is cloudy. Try again.");
        }
    } catch (UncertainFutureException e) {
        // Gracefully handling the uncertainty of future predictions
        return "Even the app is puzzled today. Maybe just do what feels right?";
    }
}
```

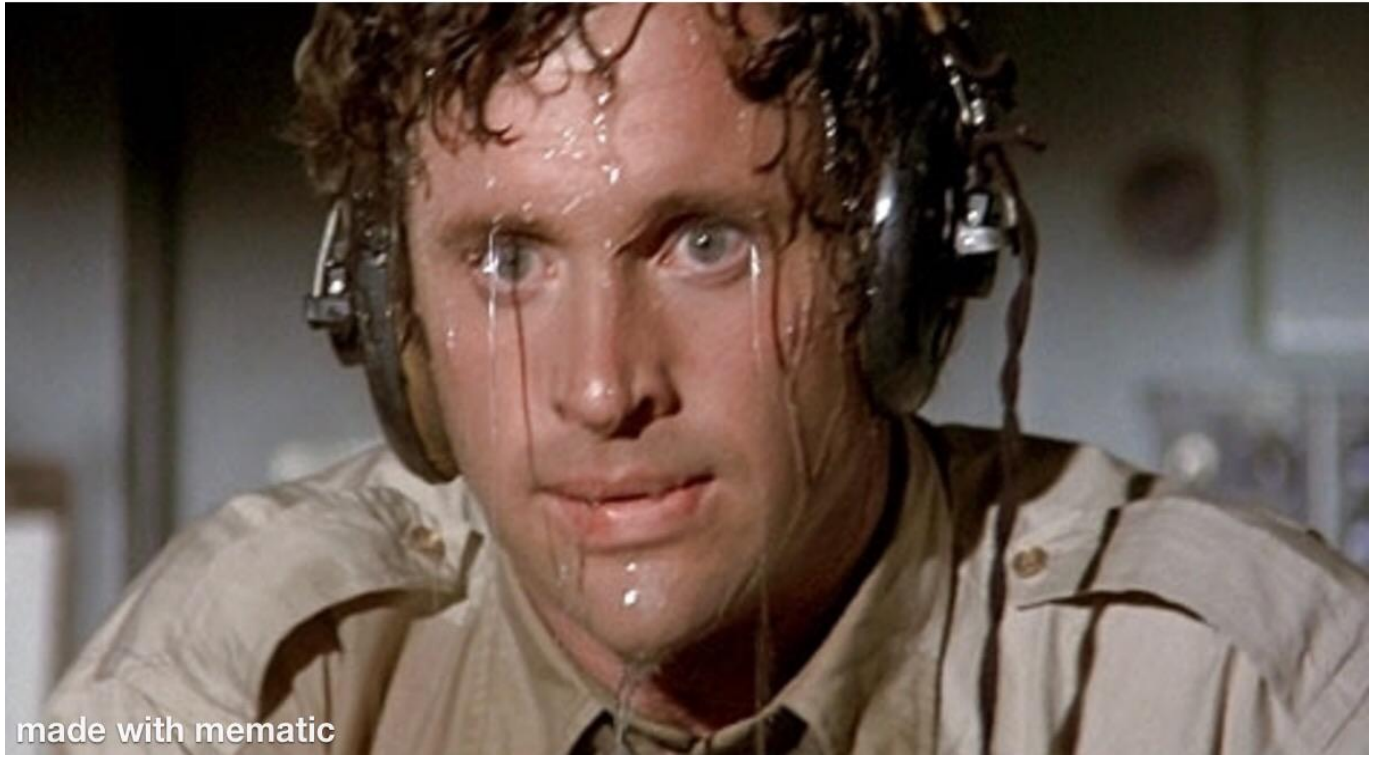
Impact: - Keeps systems stable and functional under various conditions, enhancing

OO Design Principles

Fundamental Design Principles

- **DRY (Don't Repeat Yourself):** Avoid duplication.
- **KISS (Keep It Simple, Stupid):** Keep the design simple and straightforward.
- **YAGNI (You Aren't Gonna Need It):** Don't implement something until it is necessary.

Teachers: your code should follow the principle of DRY: Don’t Repeat Yourself
My code:



DRY Principle

Definition: Avoid duplication in code. Every piece of knowledge should have a single, unambiguous representation in the system.

Impact:

- Reduces redundancy, making code easier to maintain and modify.

DRY (Don’t Repeat Yourself) Principle

```
// Pre-DRY: The "Echo" Code
class MealPlanner {
    void planBreakfast() {
        print("Brew coffee");
        print("Toast bagel");
        // More breakfast planning
    }
    void planLunch() {
        print("Brew coffee");
        print("Make sandwich");
        // More lunch planning
    }
}
```

```
// Post-DRY: The "Harmony" Code
class MealPlanner {
    void brewCoffee() {
        print("Brew coffee");
    }
    void planBreakfast() {
        brewCoffee();
        print("Toast bagel");
        // More breakfast planning
    }
    void planLunch() {
        brewCoffee();
        print("Make sandwich");
        // More lunch planning
    }
}
```

KISS Principle

Definition: Simplicity in design. Avoid unnecessary complexity and keep things straightforward.

Impact:

- Enhances understandability and reduces the chance of errors. Simplified code is often more reliable and easier to debug.


MINDY KALING
PAUL LIEBERSTEIN
Whenever I'm about to do something,
I think, "Would an idiot do that?"
And if they would, I do not do that thing.

KISS Principle

Pre-KISS

```java
// Pre-KISS: "Mission Control"
class BeverageDispenser {
    String selectBeverage(int buttonPressed) {
        // Check if it's a leap year to decide on the extra espresso shot
        if ((buttonPressed == 1) && ((Year.now().getValue() % 4) == 0)) {
            return "Extra espresso shot, because leap year!";
        } else if (buttonPressed == 2) {
            // Calculate the gravitational pull of the moon to adjust sugar levels
            return "Sugar adjusted for the moon's pull";
        }
        return "Beverage selected";
    }
}
```

Post-KISS

```java
// Post-KISS: "Just Press Play"
class BeverageDispenser {
    String selectBeverage(int buttonPressed) {
        // Simply dispense the chosen beverage
        switch (buttonPressed) {
            case 1: return "Espresso, straight up!";
            case 2: return "Sugar? We got you, just as you like!";
            default: return "Beverage selected, enjoy!";
        }
    }
}
```

YAGNI Principle

Definition: Implement only those features that are necessary. Avoid adding functionality until it is required.

Impact:

- Prevents over-engineering and keeps the codebase lean and focused. Ensures resources are used effectively on what’s truly needed.

YAGNI Principle

```
// Pre-YAGNI: The "Just In Case" Backpack
class PartyPlanner {
    void planMassiveFeast() { /* Code for an epic feast */ }
    void hireLiveBand() { /* Maybe we'll have a dance floor? */ }
    void reserveSpaceForElephant() { /* Because why not? */ }
}
```

```
// Post-YAGNI: The "Actually Needed" Kit
class PartyPlanner {
    void planMassiveFeast() { /* Still here, because, food. */ }
    // Removed the live band and elephant reservations.
    // Turns out, not every party needs an elephant.
}
```

More Design Principles

- **Separation of Concerns:** Different functionality managed by separate code.
- **Principle of Least Astonishment:** Software behaves how users will expect it to.
- **Law of Demeter:** Object should assume as little as possible about the structure or properties of anything else.

Separation of Concerns

Definition:

- Divide a program into distinct sections, each handling a specific aspect of the application's functionality.

Impact:

- Helps in isolating issues, streamlining updates, and collaborate more effectively.

Separation of Concerns

Before:

```
// Before: Mixing data access with business logic
class UserHandler {
    void createUser(String username) {
        // Database connection code
        // User creation logic
    }
}
```

After:

```

// After: Separated data access and business logic
class UserDataAccess {
    void saveUser(User user) { /* Database code to save user */ }
}
class UserHandler {
    UserDataAccess dataAccess = new UserDataAccess();
    void createUser(String username) {
        User user = new User(username);
        dataAccess.saveUser(user); // Separated concern
    }
```

Separation of Concerns

Before:

```
// Pre-Separation: The "Cluttered Kitchen" Approach
class MealPreparation {
    void prepareMeal(String ingredient) {
        // Look for ingredients in the fridge
        // Mix ingredients in a bowl
        // Cook the mixed ingredients
    }
}
```

After:

```

// Post-Separation: The "Organized Chef" Method
class IngredientStorage {
    String fetchIngredient(String name) { /* Retrieve from storage */ }
}
class Mixer {
    Bowl mixIngredients(List<String> ingredients) { /* Mixing logic */ }
}
class Stove {
    Meal cook(Bowl bowl) { /* Cooking logic */ }
}
class MealPreparation {
    IngredientStorage storage = new IngredientStorage();
    Mixer mixer = new Mixer();
    Stove stove = new Stove();

    void prepareMeal(String ingredientName) {
        String ingredient = storage.fetchIngredient(ingredientName);
        Bowl mixed = mixer.mixIngredients(Arrays.asList(ingredient));
        Meal meal = stove.cook(mixed); // Cleanly separated concerns
```

Principle of Least Astonishment

Definition:

- Software should behave in a way that users predictably expect. The design should match common user expectations to prevent confusion and errors.

Impact:

- Enhances user satisfaction and system intuitiveness.

Principle of Least Astonishment

Before:

```
// Before: Confusing method name
class FileProcessor {
    void erase(File file) { /* Actually saves the file */ }
}
```

After:

```
// After: Method name reflects its action
class FileProcessor {
    void save(File file) { /* Clearly saves the file */ }
}
```

Law of Demeter

Definition:

- Minimal knowledge between objects. An object should only interact with its direct components and not concern itself with the internal details of other objects.

Impact:

- Reduces the dependencies between components of a system, leading to a looser coupling and more modular architecture.

Law of Demeter

Before:

```
// Before: Violating Law of Demeter
class Customer {
    Wallet wallet;
    //...
}
class Shop {
    void chargeCustomer(Customer customer) {
        double amount = customer.wallet.getMoney(); // Directly accessing customer's wallet
        // Charge logic
    }
}
```

After:

```
// After: Adhering to Law of Demeter
class Customer {
    Wallet wallet;
    double payAmount(double amount) { return wallet.deduct(amount); }
}
class Shop {
    void chargeCustomer(Customer customer) {
        customer.payAmount(50); // Interacting only with the Customer interface
    }
}
```

GRASP Principles

GRASP Principles - Core Concepts

What is GRASP?

General Responsibility Assignment Software Patterns (GRASP) are guidelines for assigning responsibilities in object-oriented design to improve robustness and maintainability.

GRASP Principles

Principles Overview:

- **Information Expert:** Responsibilities go to the class with the most related information.
- **Creator:** The class that needs an object or has initializing data should create it.
- **Controller:** Designate a class to handle system events and user input.
- **Low Coupling:** Minimize class interdependencies for flexibility.
- **High Cohesion:** Keep related functions together for focused class design.
- **Polymorphism:** Handle alternatives based on object types dynamically.
- **Pure Fabrication:** Create classes for better design, even if they don’t represent real-world entities.

Information Expert

Principle: Assign responsibility to the class that has the necessary information to fulfill it.

- Classes should be assigned responsibilities based on the data they hold or the information they can access.

Information Expert

Benefits:

- Reduces redundancy and complexity.
- Enhances maintainability and cohesion.
- Makes the system more modular and easier to navigate.

Application:

- Typically applied during the design phase to ensure that each class has a clear and focused role, handling operations that are directly related to its information.

Information Expert - Before

```java
class Pizza {
    private List<String> toppings;
    // Other pizza-related methods...
}

class PizzaPriceCalculator {
    // This class eagerly jumps in to calculate the pizza price
    double calculatePrice(Pizza pizza) {
        double price = 10; // base price
        for(String topping : pizza.getToppings()) {
            price += 0.50; // Assuming each topping adds 50 cents
        }
        return price;
    }
}
```

- `OrderCalculator` is taking the responsibility that naturally belongs to the `Order` class, leading to a less intuitive and more fragmented design.

Information Expert - After

- Assign the responsibility of calculating the total cost to the class with the most knowledge required to perform it - the Order class.

```java
class Order {
    private List<Item> items;

    // Information Expert for calculating total cost
    double calculateTotalCost() {
        double total = 0;
        for(Item item : items) {
            total += item.getPrice();
        }
        return total;
    }
}

class OrderCalculator {
    // No longer responsible for calculating total cost
}
```

Creator - Overview

Principle: Assign class B the responsibility to create an instance of class A if B closely uses A or holds the data that will initialize A.

- Ensure that objects are created by the classes that use them most or have the necessary data to initialize them.

Creator - Overview

Benefits:

- Enhances encapsulation and clarity.
- Reduces dependencies and coupling between classes.
- Simplifies the system's structure.

Application:

- Useful in deciding where to put creation logic, especially in complex systems where the right placement of object creation can significantly affect the design's clarity and maintainability.

Creator - Before

```java
// Order is responsible for creating Item instances,
// but it doesn't directly contain or closely use them

class Order {
    void addNewItem() {
        Item newItem = new Item(); // Order creates Item instances
        // ...
    }
}

class ShoppingCart {
    private List<Item> items;
    // ShoppingCart logic...
}
```

Problem:

- The `Order` class is creating `Item` instances, but it doesn't have a logical or direct relationship with `Item`, leading to poor encapsulation and design.

Creator - After Applying

- Assign the responsibility to create `Item` instances to the `ShoppingCart` class, which logically contains and manages items.

```
class Order {
    // No longer responsible for creating Item instances
}

class ShoppingCart {
    private List<Item> items;

    // Creator for Item instances
    void addItem() {
        Item newItem = new Item(); // ShoppingCart creates Item instances
        items.add(newItem);
    }
}
```

Controller - Overview

Principle: Assign the responsibility of handling a system event to a class representing the overall system, a root object, or a subsystem.

- Think of it as having a dedicated barista (controller) who takes your order (input) and communicates it to the kitchen (system logic), instead of you shouting your order directly at the chefs.

Controller - Overview

Benefits:

- Centralizes control logic.
- Decouples the UI from the underlying business logic.
- Simplifies maintenance and enhances scalability.

Application:

- Typically used to define how the system will respond to user actions or other events, ensuring there’s a clear and consistent way to manage these interactions.

Controller - Before Applying

```
class UserInterface {
    // Directly handling user input and business logic
    void onLoginButtonClick(String username, String password) {
        // Validate credentials
        // Directly access the database to verify user
        // Manage session
    }
}
```

- The `UserInterface` class is overloaded with responsibilities, handling UI events, business logic, and data access, making the system hard to maintain and scale.

Controller - After Applying

Solution:

- Introduce a controller class to act as an intermediary between the UI and the system logic.

```
class UserInterface {
    // Delegates handling of the login event to the LoginController
    void onLoginButtonClick(String username, String password) {
        LoginController controller = new LoginController();
        controller.handleLoginRequest(username, password);
    }
}

class LoginController {
    AuthenticationService authService;

    void handleLoginRequest(String username, String password) {
        // Handle the login process
        User user = authService.authenticate(username, password);
        // Manage session and other login-related tasks
    }
}
```

Low Coupling

Understanding Coupling

Definition:

- Coupling is the measure of how interdependent classes or modules are. Low coupling means that a change in one class has minimal impact on other classes.

Understanding Coupling

```
// High Coupling: The Blender-Oven-Light Fiasco
class Blender {
    Oven oven;
    void blend() {
        oven.preheat(); // Why does blending require the oven?
        // ...blend something
    }
}
```

Low Coupling

Principle: Design to minimize the dependencies between classes to reduce the impact of changes and improve reusability.

- Low Coupling involves reducing the interconnectedness of classes so that changes in one class have minimal impacts on others.

Low Coupling

Benefits:

- Increases the flexibility of the system.
- Makes the codebase more resilient to changes.
- Facilitates easier testing and maintenance.

Application:

- Critical in designing systems where change is anticipated, ensuring that modifications in one part of the system don’t cause widespread issues.

Low Coupling - Before Applying

```
class Toaster {
    // Directly wired to the coffee maker
    CoffeeMaker coffeeMaker;

    void toast() {
        coffeeMaker.brew(); // Toasting bread shouldn't brew coffee
        // ...toast bread
    }
}
```

- The Toaster is overly attached to the CoffeeMaker, leading to unexpected breakfast results.

Low Coupling - After Applying

Solution:

- Impact: Now, the Toaster and CoffeeMaker can operate without causing breakfast bedlam, thanks to their newfound independence through the KitchenManager (and Appliance interface).

```
interface Appliance {
    void activate();
}

class Toaster implements Appliance {
    void activate() { /* Toast bread */ }
}

class CoffeeMaker implements Appliance {
    void activate() { /* Brew coffee */ }
}

class KitchenManager {
    Appliance appliance;

    void useAppliance() {
        appliance.activate(); // Chooses which appliance to use, independently
    }
}
```

High Cohesion

Understanding Cohesion

Definition: - Cohesion refers to the degree to which elements of a module or class belong together.

- A class with high cohesion performs a small range of tasks related to a particular purpose or concept.

Understanding Cohesion

```
1 // Low Cohesion: Handles unrelated tasks
2 class Utility {
3     void handleDataProcessing() { /*...*/ }
4     void manageUserInterface() { /*...*/ }
5 }
```

High Cohesion - Overview

Principle: Keep related and similar functionalities together in a class, ensuring that each class has a clear, narrowly focused role.

- Classes should not take on responsibilities that could be better handled by others.

High Cohesion - Overview

Benefits:

- Simplifies understanding of the system.
- Enhances the ability to manage and modify code.
- Promotes single responsibility and focused class design.

Application: - Essential in ensuring that the system remains organized and each part is as independent and focused as possible, promoting better design and easier future changes.

High Cohesion - Before Applying

```
class UserManager {
    void createUser() { /*...*/ }
    void deleteUser() { /*...*/ }
    void generateReport() { /* Unrelated to user management */ }
}
```

- `UserManager` is handling both user management and report generation, making it less cohesive and more complex.

High Cohesion - After Applying

```
class UserManager {
    void createUser() { /*...*/ }
    void deleteUser() { /*...*/ }
    // Removed report generation
}

class ReportGenerator {
    void generateReport() { /* Focused on report generation */ }
}
```

- Higher cohesion in `UserManager` and `ReportGenerator` makes each class more focused, understandable, and maintainable.

Polymorphism - Overview

Principle: Use polymorphism to handle alternatives based on object type, where the behavior varies depending on the class of the object.

- Polymorphism in object-oriented programming allows objects of different classes to be treated as objects of a common superclass. It’s a way to use a single interface to represent different underlying forms (data types).

Polymorphism

Benefits:

- Enhances flexibility and reusability by allowing different classes to be used interchangeably.
- Simplifies code by eliminating the need for multiple conditional statements.

Application:

- Commonly used when implementing system behaviors that can vary across different classes but are accessed through a common interface.

Polymorphism - Before Applying

```java
class AnimalSound {
    void makeSound(Animal animal) {
        if (animal instanceof Dog) {
            System.out.println("Woof");
        } else if (animal instanceof Cat) {
            System.out.println("Meow");
        }
        // More conditions for other animal types
    }
}

class Dog extends Animal { /*...*/ }
class Cat extends Animal { /*...*/ }
```

- `AnimalSound` class becomes cumbersome and difficult to maintain as more animal types are added.

Polymorphism - After Applying

```java
abstract class Animal {
    abstract void makeSound();
}

class Dog extends Animal {
    void makeSound() { System.out.println("Woof"); }
}

class Cat extends Animal {
    void makeSound() { System.out.println("Meow"); }
}

class AnimalSound {
    void makeSound(Animal animal) {
        animal.makeSound(); // Polymorphism in action
    }
}
```

- Each animal class knows how to make its sound, eliminating the need for conditional logic in `AnimalSound` and making the code

Pure Fabrication - Overview

Principle: Create a class that doesn't represent a concept in the problem domain, particularly to achieve low coupling, high cohesion, or to encapsulate change.

- Pure Fabrication is a made-up class that doesn't represent anything in the real world!

Pure Fabrication

Benefits:

- Enhances maintainability and reusability.
- Allows for better separation of concerns and encapsulation.

Application:

- Useful when a behavior doesn’t fit well into existing real-world domain classes or when a particular design problem is best solved independently of the domain model.

Pure Fabrication - Before Applying

```
1 class Order {
2     // Order related data and methods...
3
4     void saveOrder() {
5         // Direct database access code to save the order
6         // This mixes business logic with data access logic
7     }
8 }
```

- The `Order` class is directly handling database operations, which is not its primary responsibility, leading to a design that's hard to maintain and scale.

Pure Fabrication - After Applying

```
class Order {
    // Order related data and methods...
}

class OrderRepository {
    void saveOrder(Order order) {
        // Specific database access code to save the order
    }
}
```

- `OrderRepository` is a pure fabrication that takes over database responsibilities, leading to a cleaner separation of concerns and a more maintainable design.

SOLID Principles

SOLID Principles - Core Concepts

What is SOLID?

SOLID represents five fundamental principles in object-oriented programming and design that promote software maintainability and extensibility.

SOLID Principles - Core Concepts

Key Objectives:

- **Single Responsibility:** Encourage classes to have one reason to change.
- **Open/Closed:** Design modules that are open for extension but closed for modification.
- **Liskov Substitution:** Ensure subclasses can replace their superclasses without altering the program's correctness.
- **Interface Segregation:** Favor client-specific interfaces over general-purpose ones.
- **Dependency Inversion:** Depend on abstractions rather than concrete implementations.

SINGLE RESPONSIBILITY PRINCIPLE

Just Because You Can, Doesn't Mean You Should

Single Responsibility Principle (SRP) - Overview

Definition:

A class should have one, and only one, reason to change, promoting modularity and separation of concerns.

Importance:

- Simplifies understanding and modification of classes.
- Enhances cohesion and reduces the impact of changes.

SRP - Before Applying

```
1 class UserManager {
2     void createUser() { /*...*/ }
3     void sendEmail(String message) { /*...*/ } // Unrelated to user management
4 }
```

- `UserManager` handling both user creation and email sending mixes different concerns, making it less cohesive and more prone to errors.

SRP - After Applying

```
1 class UserManager {
2     void createUser() { /*...*/ }
3 }
4
5 class EmailService {
6     void sendEmail(String message) { /*...*/ }
7 }
```

- Each class now has a single responsibility, making the system more maintainable and understandable.

OPEN CLOSED PRINCIPLE

Open Chest Surgery Is Not Needed When Putting On A Coat

Open/Closed Principle (OCP) - Overview

Definition:

Software entities should be open for extension but closed for modification, promoting flexible and scalable systems.

- Use abstraction and polymorphism to extend behavior without altering existing code.

Open/Closed Principle (OCP) - Overview

How?:

- Designing a class hierarchy where new functionality can be added through subclasses or implementing interfaces without changing existing code.

OCP - Before Applying

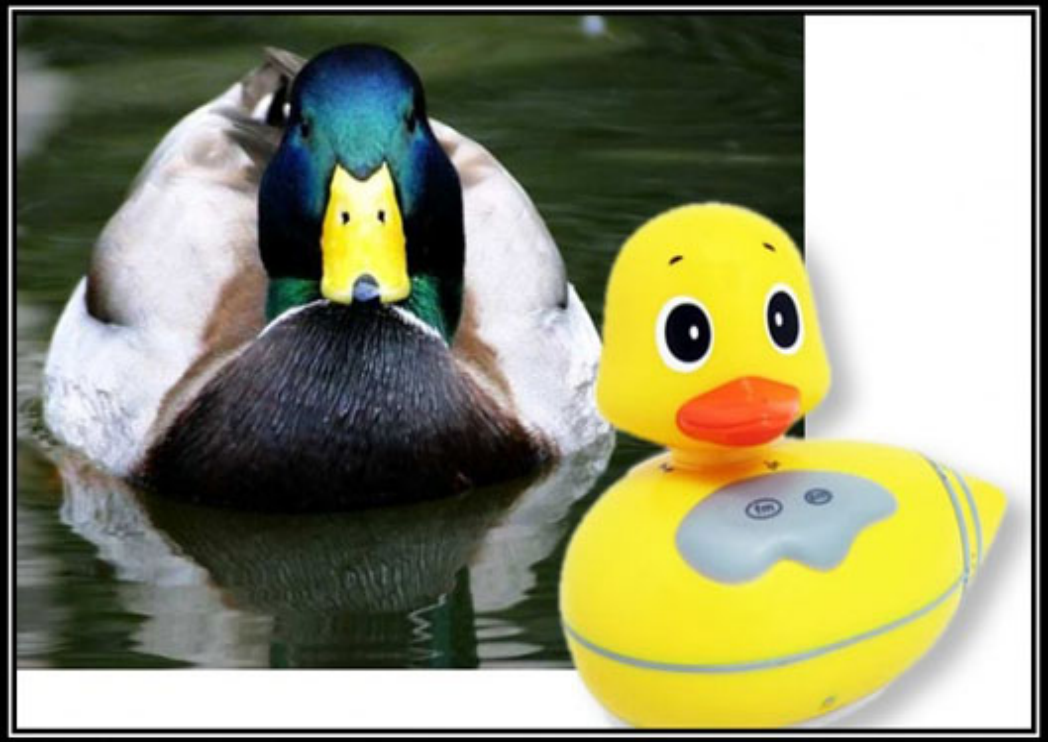
```
1  class GraphicEditor {
2      void drawShape(Shape shape) {
3          if (shape.type == 1) {
4              drawRectangle(shape);
5          } else if (shape.type == 2) {
6              drawCircle(shape);
7          }
8          // Adding a new shape requires modifying this method
9      }
10 }
```

- Adding a new shape requires changes to `GraphicEditor`, violating the open/closed principle.

OCP - After Applying

```
1  abstract class Shape {
2      abstract void draw();
3  }
4
5  class Rectangle extends Shape {
6      void draw() { /* Draw rectangle */ }
7  }
8
9  class Circle extends Shape {
10     void draw() { /* Draw circle */ }
11 }
12
13 class GraphicEditor {
14     void drawShape(Shape shape) {
15         shape.draw(); // No modification needed for new shapes
16     }
17 }
```

- New shapes can be added without modifying `GraphicEditor`, adhering to the open/closed principle and making the system more



LISKOV SUBSTITUTION PRINCIPLE

If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

Liskov Substitution Principle (LSP) - Overview

Definition:

Subclasses should be substitutable for their base classes without affecting the program's correctness.

- Ensures that a subclass can stand in for its parent class without causing unexpected behavior.

LSP - Before Applying

Java Example:

```java
class Bird {
    void fly() { /*...*/ }
}

class Ostrich extends Bird {
    // Ostriches can't fly, but they're subclassed from Bird
    void fly() { throw new UnsupportedOperationException(); }
}
```

- Using an `Ostrich` object where a `Bird` is expected can cause unexpected errors due to the overridden `fly` method.

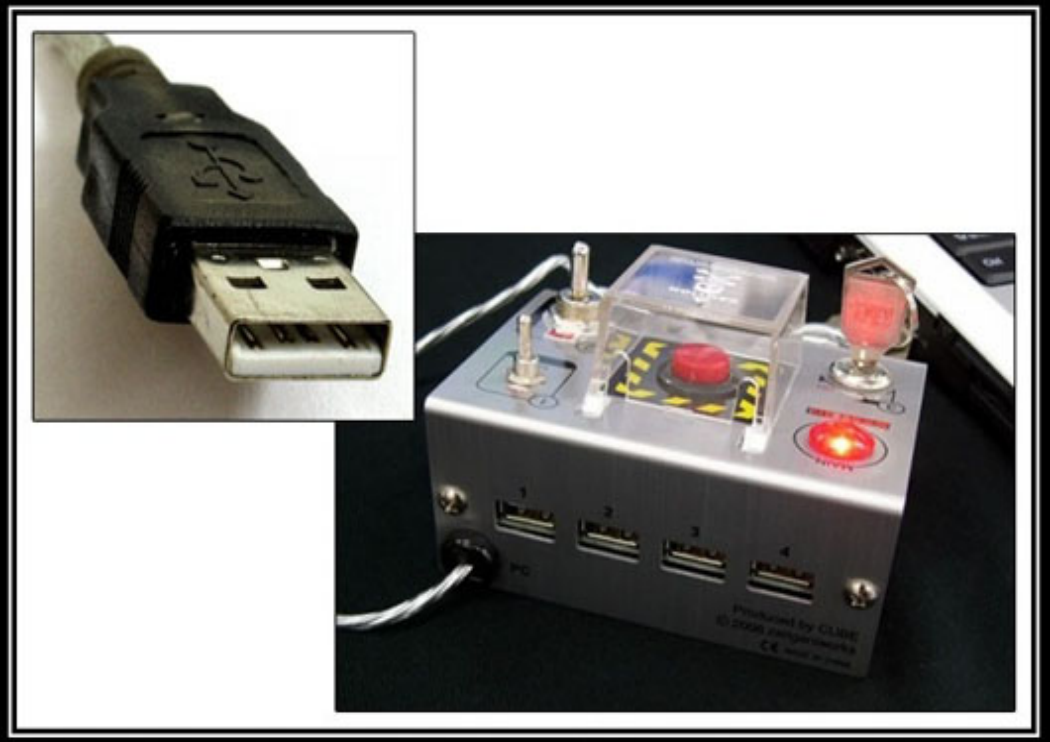
LSP - After Applying

```
abstract class Bird {
    // Some common bird behavior
}

class FlyingBird extends Bird {
    void fly() { /* Implement flying */ }
}

class Ostrich extends Bird {
    // Ostrich-specific behavior, no fly method
}
```

- `FlyingBird` and `Ostrich` are now both substitutable for `Bird` without causing unexpected behavior, adhering to the Liskov Substitution Principle.



INTERFACE SEGREGATION PRINCIPLE

You Want Me To Plug This In, Where?

Interface Segregation Principle (ISP) - Overview

Definition:

Clients should not be forced to depend on interfaces they do not use, encouraging fine-grained interfaces over general-purpose ones.

- Designing small, focused interfaces that classes can implement without being forced to include unnecessary methods.

ISP - Before Applying

```
1  interface Worker {
2      void work();
3      void eat();
4      void takeBreak();
5  }
6
7  class Robot implements Worker {
8      void work() { /*...*/ }
9      void eat() { /* Robots don't eat */ }
10     void takeBreak() { /* Robots don't take breaks */ }
11 }
```

- `Robot` is forced to implement `eat` and `takeBreak`, which are irrelevant to its functionality.

ISP - After Applying

```java
interface Worker {
    void work();
}

interface HumanWorker extends Worker {
    void eat();
    void takeBreak();
}

class Robot implements Worker {
    void work() { /*...*/ }
}

class Human implements HumanWorker {
    void work() { /*...*/ }
    void eat() { /*...*/ }
    void takeBreak() { /*...*/ }
}
```

- `Robot` now only implements the relevant `Worker` interface, and `Human` implements the extended `HumanWorker` interface. This segregation makes the system more flexible and maintainable.

Dependency Inversion Principle

Would you solder a lamp directly
to the electrical wiring in a wall?

Dependency Inversion Principle (DIP) - Overview

Definition:

High-level modules should not depend on low-level modules; both should depend on abstractions. Abstractions should not depend on details; details should depend on abstractions.

- Using interfaces or abstract classes to define high-level policies, which are then implemented by concrete classes without the high-level modules knowing the details of the implementation.

DIP - Before Applying

```
class OrderProcessor {
    MySQLDatabase database; // Directly dependent on a specific database implementation

    void processOrder(Order order) {
        database.save(order); // Tightly coupled to MySQLDatabase
    }
}
```

- `OrderProcessor` is directly dependent on `MySQLDatabase`, making it hard to switch to a different database or test the order processing independently.

DIP - After Applying

```
interface Database {
    void save(Order order);
}

class MySQLDatabase implements Database {
    void save(Order order) { /*...*/ }
}

class OrderProcessor {
    Database database; // Depends on the abstraction

    void processOrder(Order order) {
        database.save(order); // Not tied to a specific implementation
    }
}
```

- `OrderProcessor` now depends on the `Database` abstraction, not the concrete `MySQLDatabase` implementation. This makes the system more flexible and easier to modify or test.

See you in the lab!