

# **Object Oriented Modelling and Design**

**Dr. Ashish Sai**

**BCS1430**

EPD150 MSM Conference Hall

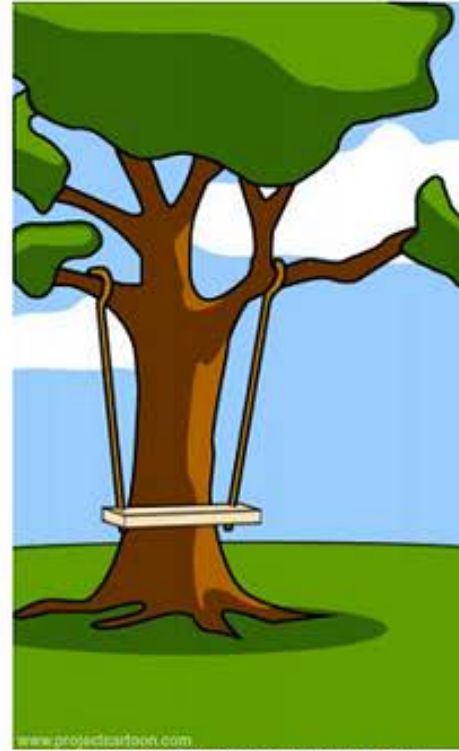
Week 2 Lecture 1 & 2

# Lecture 1

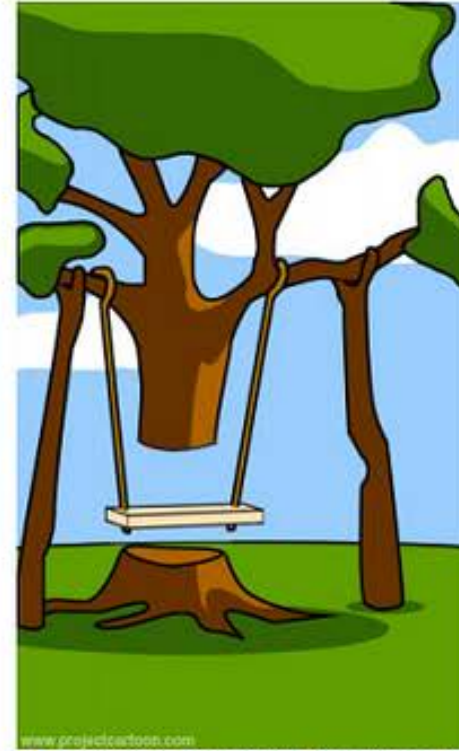
| Understanding what needs to be  
designed



How the customer explained it



How the project leader understood it



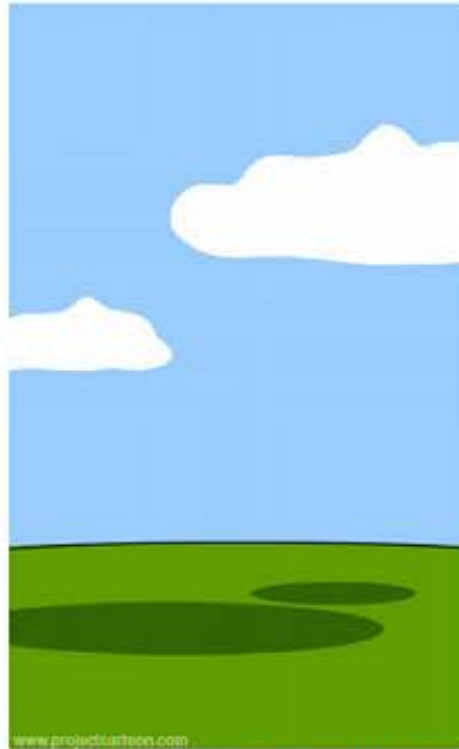
How the analyst designed it



How the programmer wrote it



How the business consultant described it



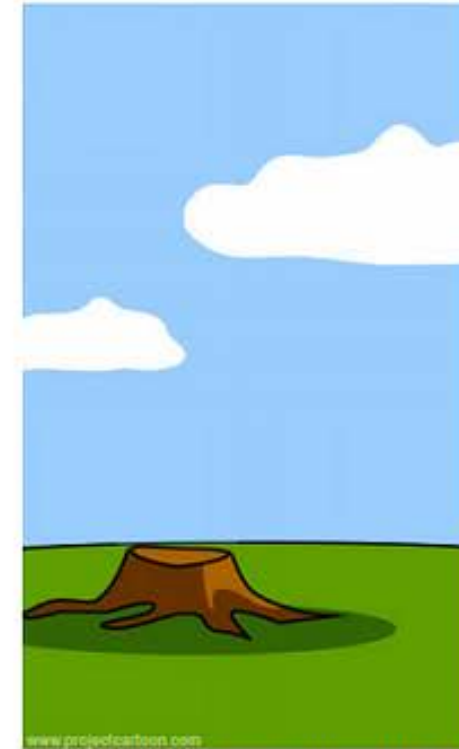
How the project was documented



What operations installed



How the customer was billed



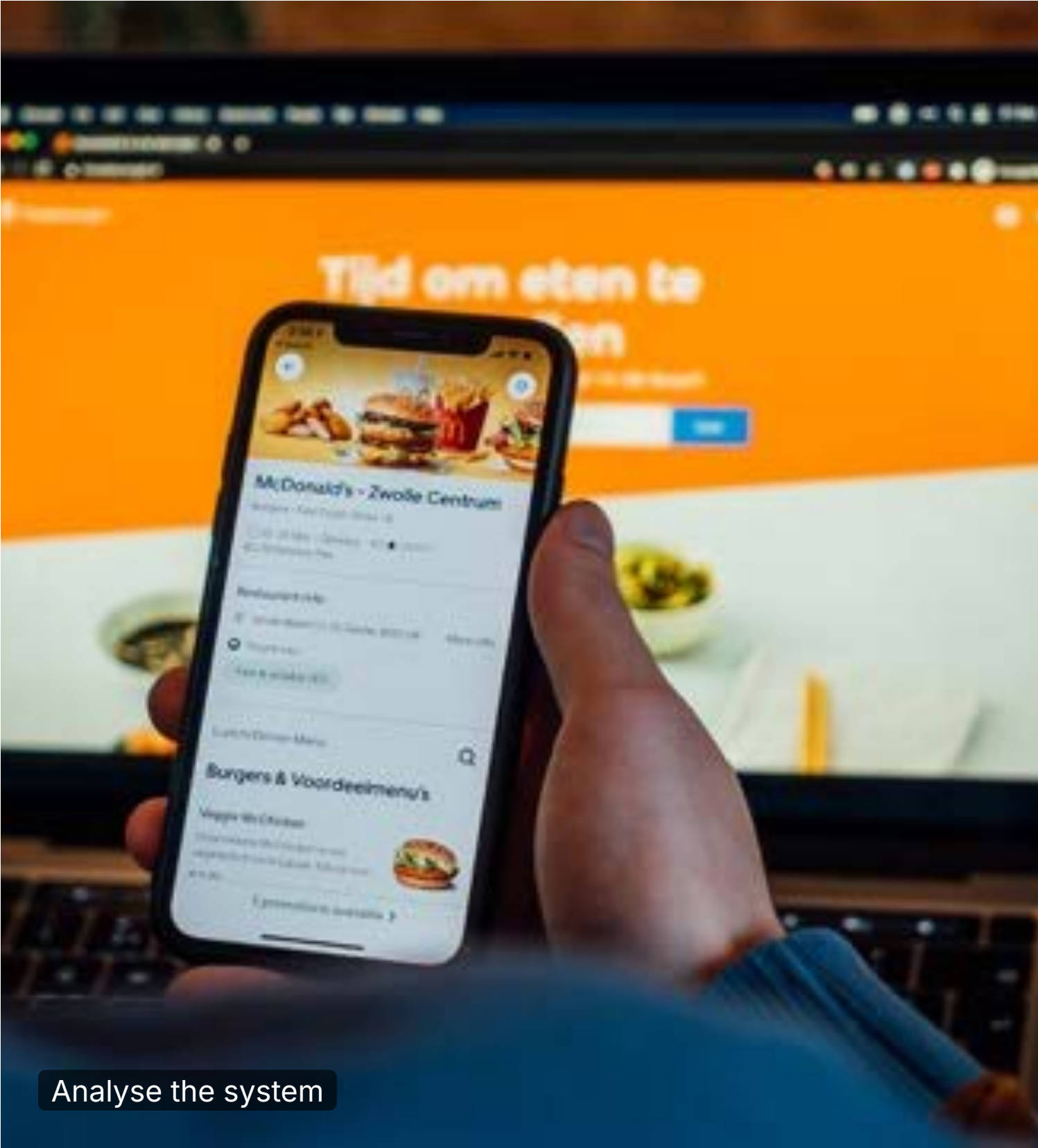
How it was supported



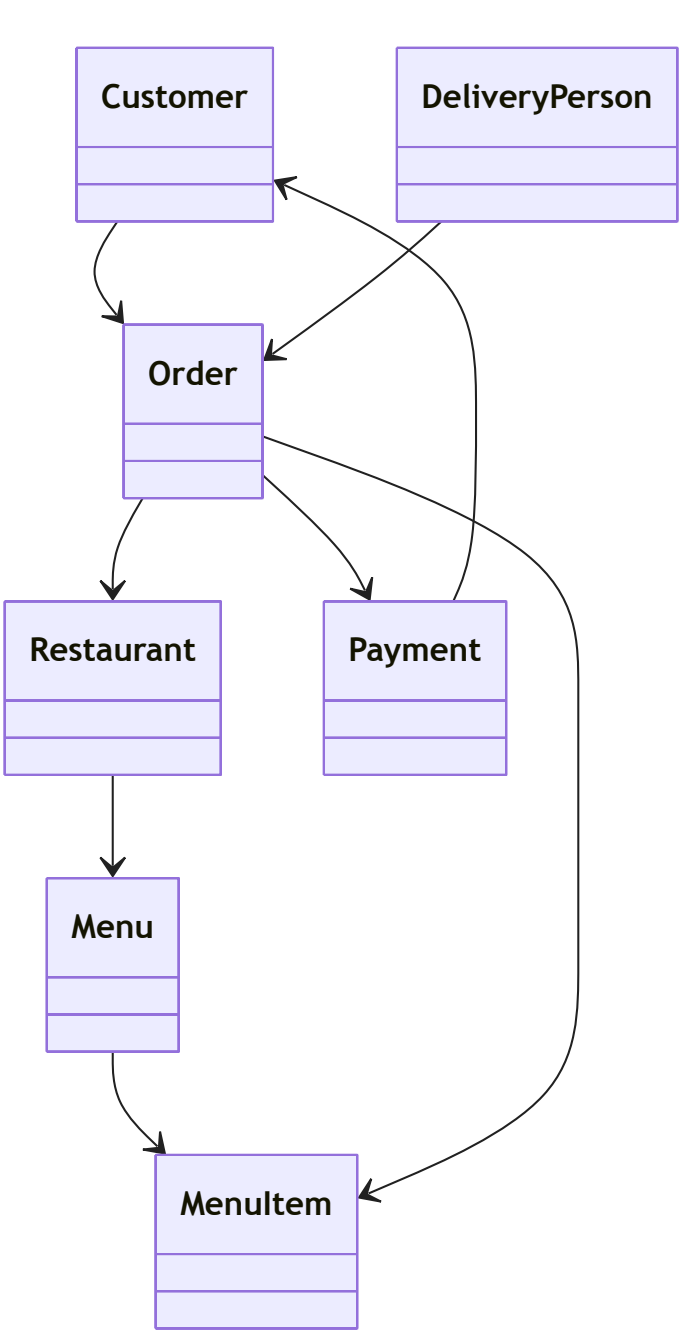
What the customer really needed

# **Object- Oriented Analysis and Design**

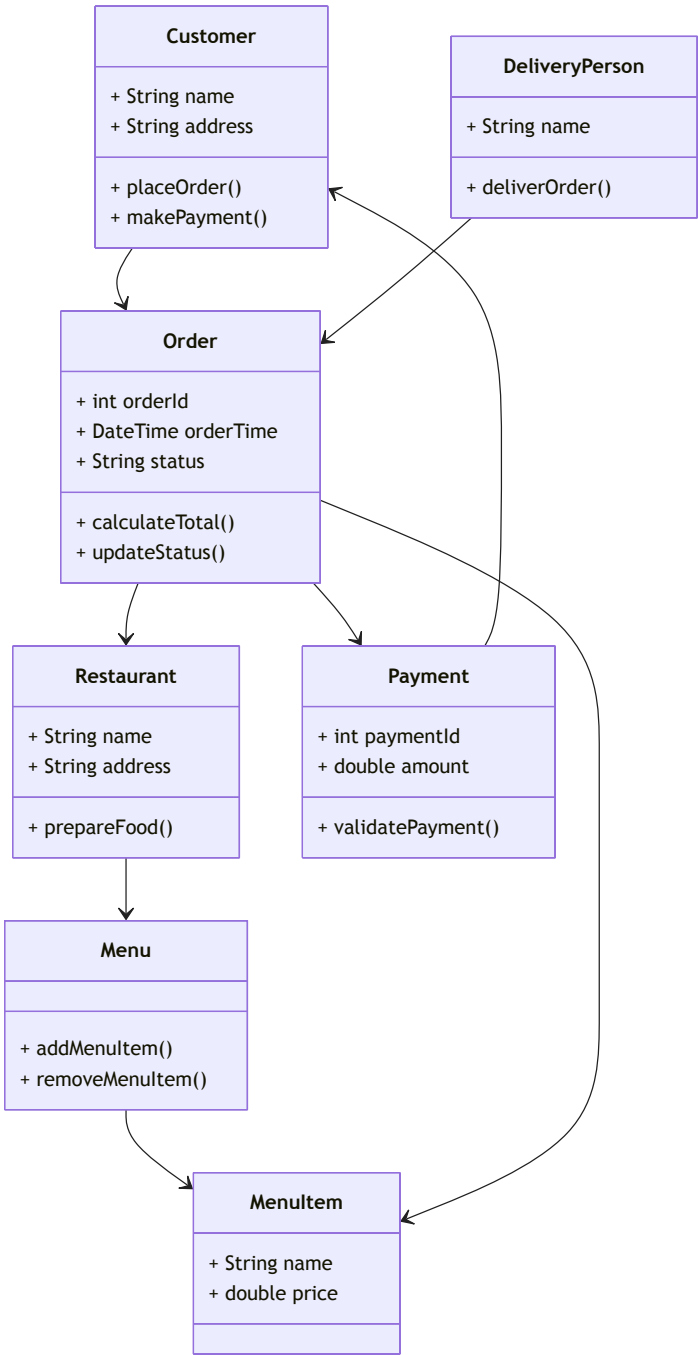




Analyse the system



Design the System



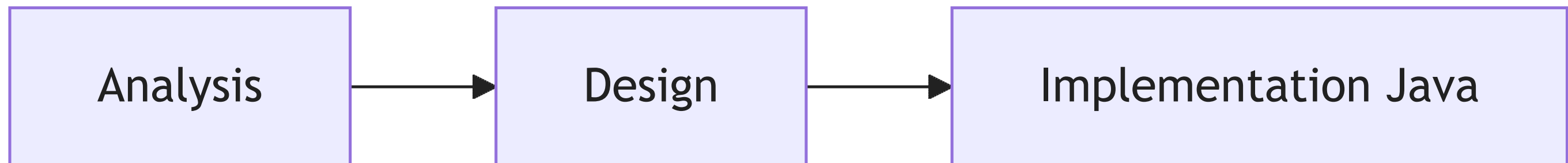
Model the system

# What is Object-Oriented Analysis and Design?

- **Analysis**: What are we trying to do?
- **Design**: How are we going to do it?

# OOA & OOD Overview

- **Object-Oriented Analysis (OOA)**
  - Examine the problem or system, identify *objects* and *interactions*.
- **Object-Oriented Design (OOD)**
  - Take the analysis results and create a conceptual solution that can be implemented (e.g., in Java).



# Why does it matter?

- Complex systems are best understood via *interactions* of simpler parts.
- OOA&D leverages this to create ***modular, reusable, and flexible software.***



## Benefits of OOA&D


- **Modularity**

“Degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components.” (ISO/IEC 2011)





# ***Approve sign-in request***

-  *Open your Microsoft Authenticator app and approve the request to sign in.*

*I can't use my Microsoft Authenticator app right now*



## Benefits of OOA&D

- **Reusability**

“Degree to which an asset can be used in more than one system.” (ISO 25010)

# Benefits of OOA&D

- **Flexibility**

Systems designed with OOA&D adapt easily to changing requirements.

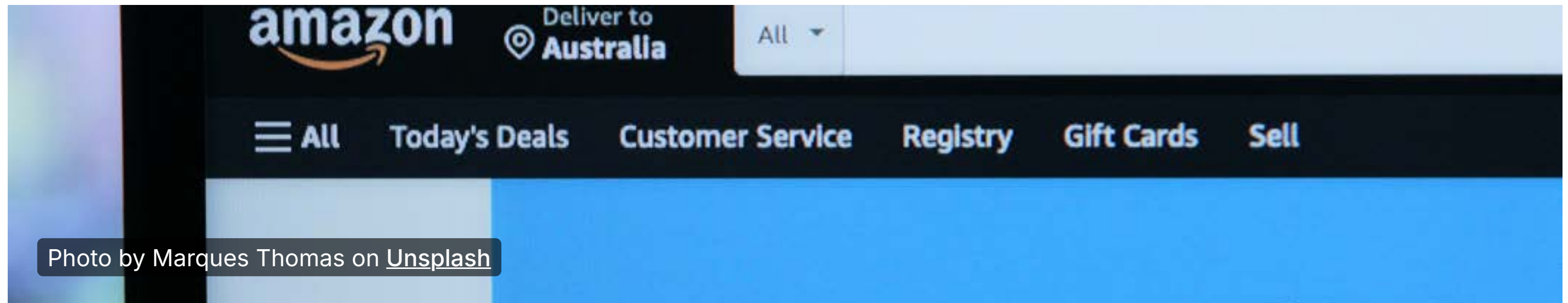


Photo by Marques Thomas on [Unsplash](#)

# OO Analysis in Software Development

- Investigative phase diving into the problem domain.
  - Identify objects and interactions.
- **Goal**: A model of the real world that can be translated into a software system.

“It's easier to change a design than to change a built system.”

# OO Design in Software Development

- Turn the conceptual model from **OOA** **into a blueprint** for building the system.
- Decide on *system architecture, components, and how they interact.*



**What is good  
software?**


# Understanding Software Quality

**ISO 25010**<sup>1</sup> defines software quality via attributes like:

- **Functionality**
- **Reliability**
- **Usability**
- **Efficiency**
- **Maintainability**
- *Portability*
- *Compatibility*
- *Security*

1. <https://www.iso.org/standard/35733.html>


# Quality Attribute: Functionality

- **Definition:** The degree to which a product or system provides functions that meet stated and implied needs.
- Focus on ***completeness, correctness, appropriateness*** of features. 

# Quality Attribute: Reliability

- **Definition:** The *ability to maintain performance* under stated conditions for a specified period.
- Concerned with maturity, fault tolerance, recoverability.

# Quality Attribute: Usability


- **Definition:** The extent to which a product can be used effectively and efficiently, with user satisfaction.
- Encompasses ***understandability***, ***learnability***, and ***operability***. 

# Quality Attribute: Efficiency

- **Definition:** The relationship between performance and resources used, under stated conditions.
- *Time behavior, resource utilization, capacity.* ⚡



# Quality Attribute: Maintainability

- **Definition:** Ease of modification to correct faults, improve performance, or adapt to changes.
- Involves **modularity, reusability, and analyzability.** 

**How to design  
good (OO)  
software?**

# Developing good (OO) Software

1. Understand the requirements
2. Analyze them in an OO context (*OO Analysis*)
3. Design the software using *OO principles*

WHO WE ARE?



CLIENTS!



WHAT DO WE WANT?



WE DON'T KNOW!



WHEN DO WE WANT IT?



NOW!



# Gathering Requirements

# Introduction to Software Requirements

- **Requirements**: Descriptions of system features and constraints.
- Essential for *guiding development, setting customer expectations, project planning.*



# **Types of Software Requirements**

- Functional Requirements**
- Non-functional Requirements**
- Domain Requirements**

# Functional Requirements (FR)

- **Describe what services the system must provide.**
- How it should respond to particular inputs or behave under certain situations.

## Examples:

- *"The system shall allow users to create an account by providing a username, password, and email address."*
- *"The system shall enable users to search and filter products by name, category, and price range."*

# Non-Functional Requirements

- Set criteria to judge the operation of a system, e.g., ***performance, reliability, usability, security.***
- **Examples:**
  - *“The system shall have an uptime of 99.9% as measured over a monthly cycle.”*
  - *“All sensitive user data shall be encrypted at rest and in transit, adhering to PCI-DSS standards.”*

# Domain Requirements

- Reflect domain-specific *knowledge, standards, or regulations*.
- **Examples:**
  - *“Store medical data in compliance with healthcare regulations.”*
  - *“Perform currency conversion per international financial standards.”*

# **The Art of Gathering Requirements**

- Collect needs and specs from stakeholders.
- Understand stakeholder goals; translate them into technical details.

# Stakeholders

- Individuals/groups with **an interest** in **project success**.
- Examples: *Clients, end-users, managers, developers, examiners.*



# Techniques for Gathering Requirements

- *Interviews, surveys, observations, workshops, brainstorming.*

# Introduction to Use Cases

- Detailed description of user tasks, focusing on **system behavior** from a **user's perspective**.
- Begins with a user's goal, ends when the goal is achieved.

# Elements of a Use Case

Element	Description
Actor	The user performing the task.
Stakeholder	Anyone with interests in the system outcome.
Preconditions	What must be true before the use case starts.
Triggers	Events initiating the use case.
Main Success Scenario	Basic flow if everything goes right.
Alternative Flows	Deviations from the main flow.

# Writing a Use Case

1. Identify **users**
2. Define **actions** (each action → a use case)
3. Describe **normal** + **alternate** courses
4. Note **commonalities**
5. **Repeat** for all users

# Simple Laundry Use Case

- **Actor**: Housekeeper
- **Basic Flow**: Sorting, washing, drying, folding, ironing, discarding.
- **Preconditions**: It's Wednesday, and there is laundry.
- **Trigger**: Dirty laundry is present.
- **Post Conditions**: All laundry is clean/folded or hung up.

# Alternative Flows

- Alternative Flow 1. If items are wrinkled, iron and hang.
- Alternative Flow 2. If still dirty, rewash.
- Alternative Flow 3. If shrunk, discard.

# Example Use Case: Enter Order



**Description:** Customer enters a new order into the system.

**Actors:** Customer, System, Market

**Preconditions:** Customer is logged on.

**Postconditions:**

- Order processed
- Customer receives confirmation

# Basic Flow of Enter Order

Step	Actor	Action	Notes
BF-1	Customer	Navigates to order entry page	
BF-2	System	Displays order entry page	
BF-3	Customer	Enters order details: Buy/Sell, Quantity, Symbol	
BF-4	Customer	Submits the order	
BF-5	System	Validates information	AF-1 if invalid
BF-6	System	Submits order to marketplace	
BF-7	Market	Executes the order	
BF-8	Market	Sends execution report to system	
BF-9	System	Displays execution report to customer	



# Alternate Flows - Invalid Info

## AF-1: Customer enters invalid information

Step	Actor	Action
AF-1-1	System	Displays error: "You have entered invalid info."
AF-1-2	Customer	Corrects info and resubmits
AF-1-3		Continue at BF-5

# Exception Flows - System Issues

## EF-1: System unavailable

Step	Actor	Action
EF-1-1	System	Displays: "System currently unavailable."
EF-1-2		Use case ends

## EF-2: Cannot connect to market

Step	Actor	Action
EF-2-1	System	Displays: "Cannot connect to the market. Please try again later."
EF-2-2		Use case ends

# Supplemental Requirements

---

ID	Name	Description
SR-1	Tabbing	Enable tabbing between fields on the order entry page.

---

# Use Cases: Key Takeaways

- Focus on **user actions** and **system responses**.
- Clearly **define** basic, alternate, and exception flows.
- Supplemental requirements address usability or user-experience details.

## Case Study: Starbucks Mobile Order & Pay

Starbucks introduced **Mobile Order & Pay** to reduce store wait times, improve efficiency, and personalize experiences.

### Objectives

1. Reduce in-store wait times
2. Improve order accuracy
3. Increase customer convenience

# Use Case: Placing an Order (Mobile)

**Actor:** Customer

**Preconditions:**

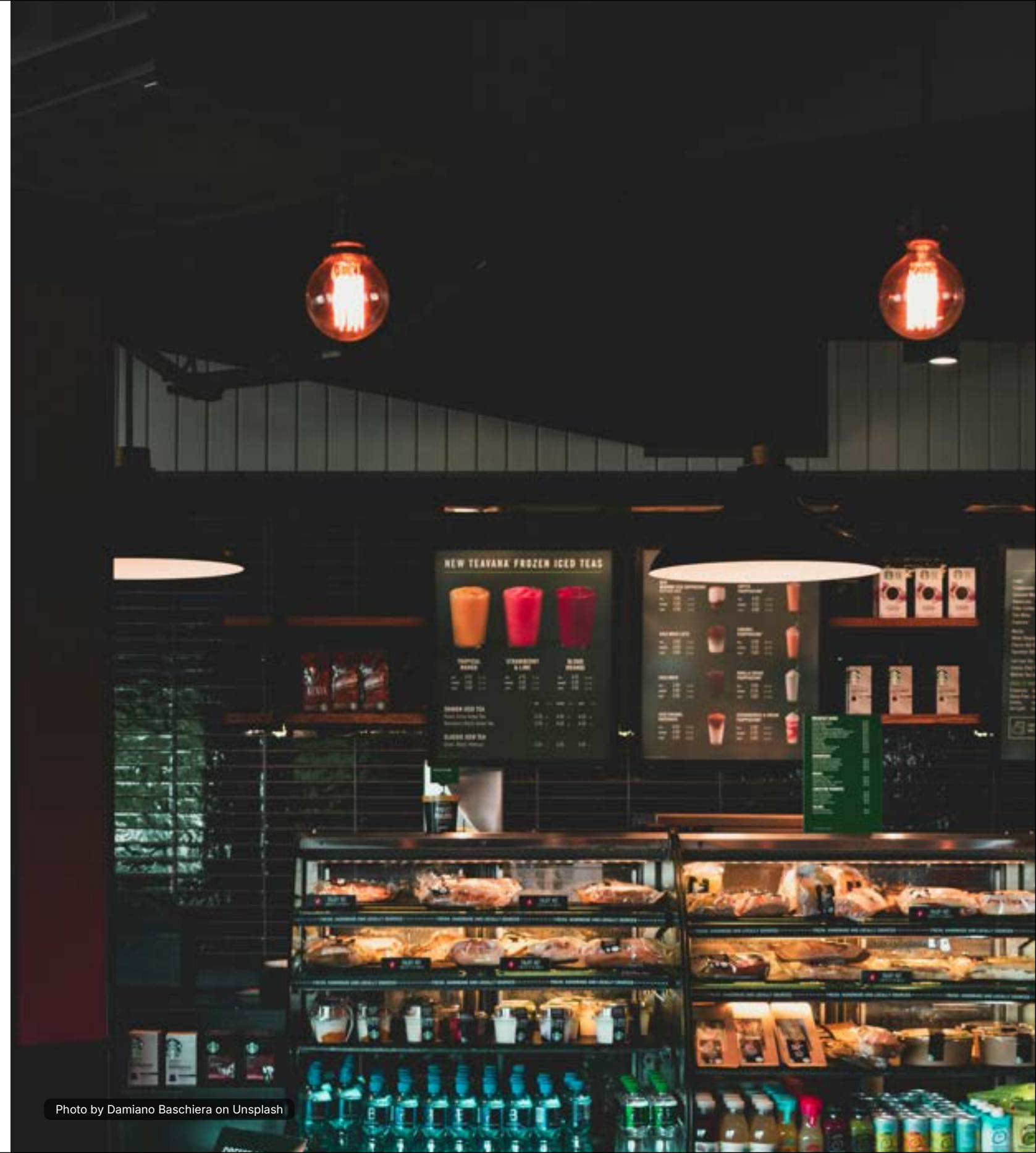
- Starbucks app installed
- User account with payment info

**Basic Flow:**

1. Customer opens app → selects "Order."
2. Browses menu and selects items.
3. Reviews order, makes modifications.
4. Confirms order, selects pickup location.
5. Pays via app.
6. Order sent to store.
7. Notification sent when order is ready.

**Postconditions:**

- Customer picks up order.



# Derived Requirements from Use Case

## Functional Requirements

- **FR1:** Allow browsing menu + selecting items
- **FR2:** Order modifications before confirmation
- **FR3:** Secure payment
- **FR4:** Send order to store + confirm readiness

## Non-Functional Requirements

- **NFR1:** Load menu within 2s
- **NFR2:** Comply with PCI DSS
- **NFR3:** Accessible & user-friendly
- **NFR4:** Handle 1000 orders simultaneously

# Requirements Prioritization

- Not all requirements are equal.
- Evaluate urgency, value, feasibility.
- **MoSCoW** (Must, Should, Could, Won't)

MoSCoW technique is a prioritization tool used to reach a common understanding on the importance of various requirements.



# MoSCoW Technique (Starbucks Example)

## Functional Requirements

- **Must (M)**
  - FR1 (browse/select items)
  - FR3 (secure payment)
- **Should (S)**
  - FR2 (order modification)
- **Could (C)**
  - Personalized recommendations
- **Won't (W)**
  - Advanced custom ordering using new tech

# MoSCoW Technique (Non-Functional)

- **Must (M)**
  - NFR2: PCI DSS compliance
  - NFR3: Basic usability
- **Should (S)**
  - NFR1: Fast menu loading (2s)
- **Could (C)**
  - NFR4: Scalability for >1000 orders
- **Won't (W)**
  - 3rd-party loyalty system integration

# **Analysis**

# Object-Oriented Analysis

- A methodical approach to understanding a system **via** **objects in the domain.**
- Key aspects:
  - Understand domain
  - Capture requirements
  - Define the system

# Identifying Objects and Classes

1. Examine the Domain
2. Spot Key Entities
3. Define Attributes/Operations

**Example:** A Library System might have `Book`, `Member`,  
`Loan`.

# Identify Relationships and Interactions

- **Associations**: "has-a" or "uses-a"
- **Aggregations**: Whole-part (less strict)
- **Compositions**: Whole-part with strong dependency

**Example:** E-commerce platform → `Customer` places an `Order` containing `Products`.

# Associations

- "has-a" or "uses-a"
- Can be one-to-one, one-to-many, many-to-one, or many-to-many.

```
class Customer {  
    private Order order;  
    void placeOrder() { /*...*/ }  
}  
  
class Order {  
    // ...  
}
```

One-to-one association: `Customer` ↔ `Order`.

# Aggregations

- Whole-part
- The part can exist independently of the whole.

```
class Team {  
    private List<Player> players;  
}  
  
class Player {  
    // ...  
}
```

A `Team` aggregates multiple `Player` objects; players can exist independently.



# Compositions

- Composition is a strong form of aggregation implying ownership.

```
class Engine {  
    // ...  
}  
  
class Car {  
    private Engine engine = new Engine();  
    void start() { /* starts engine */ }  
}
```

A `Car` strongly composes an `Engine`; if `Car` is destroyed, so is `Engine`.

# Object Interaction Analysis

- Understand how objects communicate.
- Interaction diagrams (sequence, collaboration) help visualize <sup>1</sup>.

**Example:** Airline reservation → how `Passenger` interacts with `Flights`, `Tickets`, `Payments`.

1. More on these diagrams in the next lecture.

# Assigning Responsibilities with CRC Cards

## Class-Responsibility-Collaborator (CRC)

- **Class**: The entity being modeled
- **Responsibility**: What the class does or knows
- **Collaborators**: Other classes it interacts with

# CRC Card Example - ShoppingCart

- **Class Name**: ShoppingCart

- **Responsibilities**

- addItem(Product)

- calculateTotal()

- removeItem(Product)

- checkout()

- **Collaborators**

- Product (items to add)

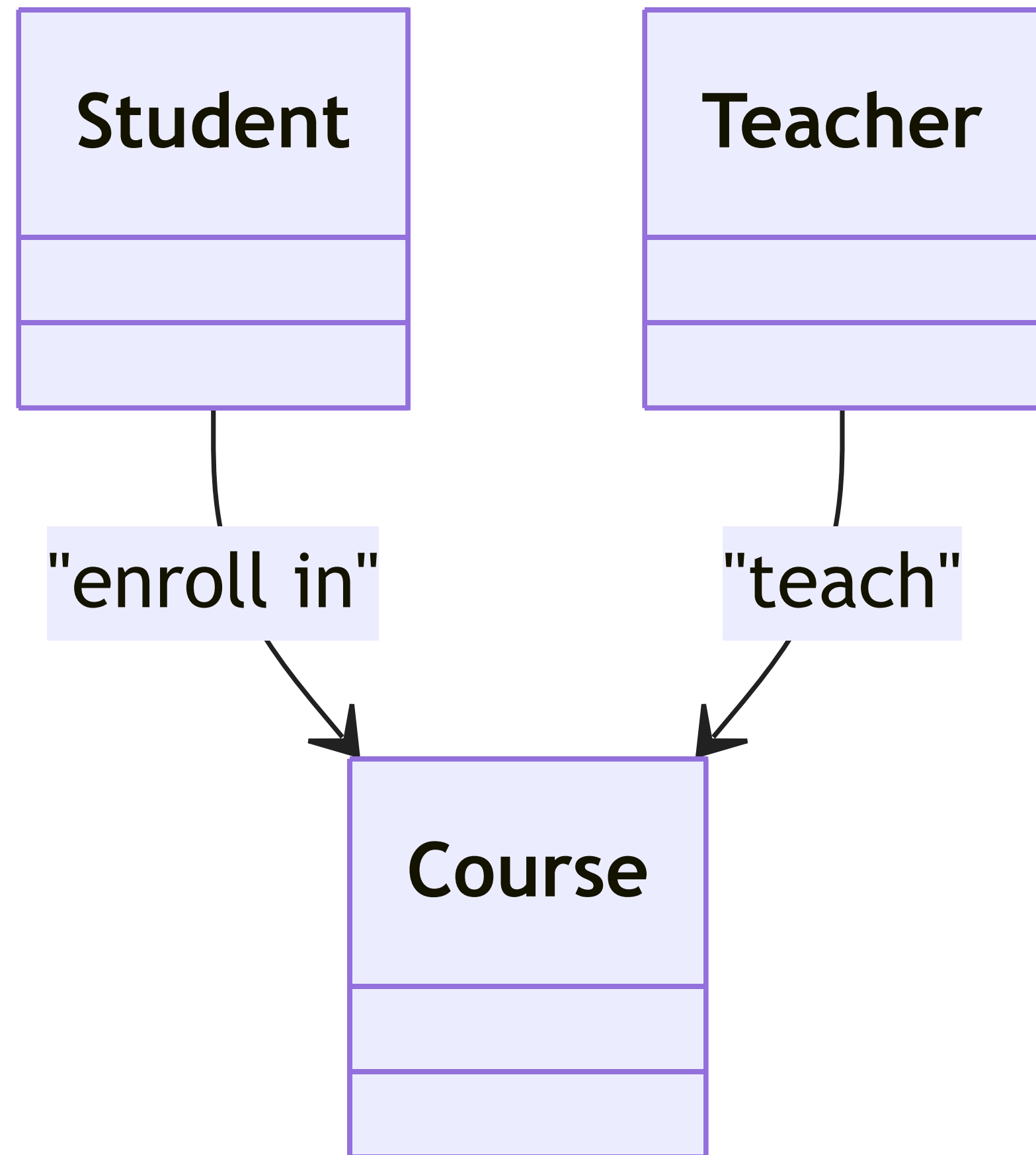
- User (cart owner)

```
class ShoppingCart {  
    private List<Product> products;  
    private User owner;  
  
    void addItem(Product product) { /*...*/ }  
    double calculateTotal() { /*...*/ }  
    void removeItem(Product product) { /*...*/ }  
    void checkout() { /*...*/ }  
}  
  
class Product { /* details */ }  
class 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.

# Class Diagrams: The Static Blueprint

**Class diagrams** show classes, their attributes, operations, and relationships.



**What is good  
design?**

# Good Design in Software

- More than aesthetics: solutions that are **effective, efficient, maintainable.**
- *Simplicity, Consistency, Modularity, Usability, Maintainability, Robustness.*



# Simplicity

- Focus on essential elements.
- Example: *A single, well-defined Java method is easier to understand/test.*

```
// 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);
        String strB = Integer.toString(b);

        // Convert strings back to integers
        int intA = convertStringToInt(strA);
        int intB = convertStringToInt(strB);

        // Perform addition in a loop
        for (int i = 0; i < intA; i++) {
            sum = increment(sum);
        }

        for (int i = 0; i < intB; i++) {
            sum = increment(sum);
        }

        // Return the calculated sum
        return sum;
    }

    // Method to convert string to integer
    private static int convertStringToInt(String number) {
        try {
            return Integer.parseInt(number);
        } catch (NumberFormatException e) {
            System.err.println("Error converting: " + e.getMessage());
            return 0;
        }
    }

    // Method to increment a number
    private static int increment(int number) {
```

# Simple:

```
int add(int a, int b) {  
    return a + b;  
}
```

Reduces complexity, enhances maintainability, and improves user experience.

# Consistency

- Uniformity in visuals, terminology, behavior.
- Example: *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

- Designing systems as separate, interchangeable components.
- **Example:** *Splitting code into Java packages*  
( `com.company.orderprocessing` vs.  
`com.company.customermanagement` ).

## Modularity

```
// 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;
public class EscapeProtocol {
    // In case someone decides to reenact a scene from a dinosaur theme park movie
}
```



```
if (submissionSuccess) {  
    displayMessage("Form  
submitted  
successfully!");  
} else {  
  
displayMessage("Submissio  
n failed. Please try  
again.");  
}
```

# Usability

- Making software **intuitive and accessible.**
- Example: *Simple, clear web app forms with immediate feedback.*

# **Maintainability ?**

**YOU DON'T HAVE TO WRITE MAINTAINABLE CODE**

**IF YOU NEVER MAINTAIN YOUR CODE**

# Maintainability

- Ease of modification and enhancement.
  - Example: *Clean, well-documented Java code adhering to patterns.*

```
/**
 * Calculates the monthly payment.
 * @param loanAmount ...
 * @param termInYears ...
 * @param interestRate ...
 */
public double calculateMonthlyPayment(...) { ... }
```

```
// 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?
    else {
        return "How about a wildcard? 'The Grand Budapest Hotel'";
    }
}
```

# Robustness

- Handling errors/unexpected situations gracefully.
  - Example: `try/catch` blocks to recover from failures.

```
try {  
    // risky operation  
} catch (Exception e) {  
    // handle + recover  
}
```

```
// "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, and a magic 8-ball
        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?";
    }
}

/**
 * Custom exception for when the future is just too murky to predict.
 */
class UncertainFutureException extends Exception {
    public UncertainFutureException(String message) {
        super(message);
    }
}
```



# Lecture 2



# OO Design Principles

# Fundamental Design Principles

- **DRY (Don't Repeat Yourself)**
- **KISS (Keep It Simple, Stupid)**
- **YAGNI (You Aren't Gonna Need It)**

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



# DRY Principle

| Avoid duplication: one unambiguous representation of info in the system.

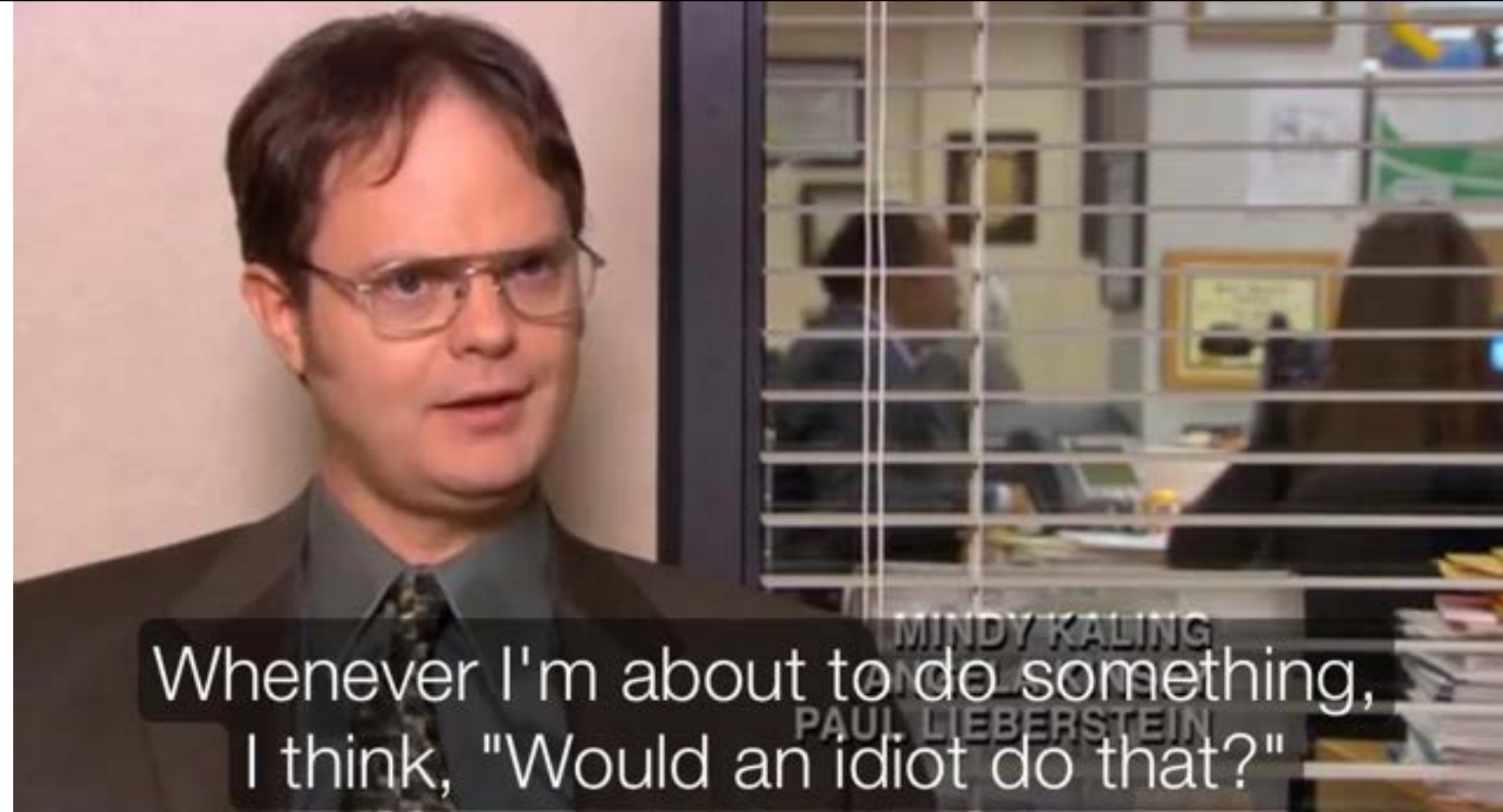
**Before:**

```
// 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
    }
}
```

**After:**

```
// 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

| Simplicity over complexity.

**Before:**

```
// 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";
    }
}
```

**After:**

```
// 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

| Implement features only when needed.

## Before:

```
// 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? */ }
}
```

## After:

```
// 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**: Each part handles a distinct aspect.
- **Principle of Least Astonishment**: Behave as users would expect.
- **Law of Demeter**: Object should know as little as possible about others.



# 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
    }
}
```

# Principle of Least Astonishment

## Before:

```
class FileProcessor {  
    void erase(File file) { /* actually saves the file */ }  
}
```

## After:

```
class FileProcessor {  
    void save(File file) { /* clearly a save method */ }  
}
```

# 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.

## Before:

```
class Shop {  
    void chargeCustomer(Customer c) {  
        double amount = c.wallet.getMoney(); // direct wallet access  
        // ...  
    }  
}
```

## After:

```
class Customer {  
    Wallet wallet;  
    double payAmount(double amt) { return wallet.deduct(amt); }  
}  
class Shop {  
    void chargeCustomer(Customer c) {  
        c.payAmount(50); // no direct wallet reference  
    }  
}
```

# **GRASP**

# **Principles**

# **General Responsibility Assignment**

**Software Patterns (GRASP)** are

guidelines for assigning responsibilities in object-oriented design to improve robustness and maintainability.

# **GRASP: Core Concepts**

General Responsibility Assignment Software Patterns:

- **Information Expert**
- **Creator**
- **Controller**
- **Low Coupling**
- **High Cohesion**
- **Polymorphism**
- **Pure Fabrication**

# Information Expert

Assign responsibility to the class that has the relevant info.

**Before:**

```
class OrderCalculator {  
    double calculateTotalCost(Order order) { ... }  
}
```

**After:**

```
class Order {  
    double calculateTotalCost() { ... }  
}
```



# **Information Expert**

## **Benefits:**

- Reduces redundancy and complexity.
- Improves maintainability, cohesion, and modularity.
- Simplifies system navigation.

## **Application:**

- Used in the design phase to assign each class a clear, focused role, managing operations tied to its own information.

# Information Expert - Before

```
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 assumes responsibilities of the Order class, resulting in a fragmented and less intuitive design.

## Information Expert - After

Assign total cost calculation to the Order class, as it holds the necessary knowledge.

```
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

The class that needs an object or has the data to initialize it is responsible for creation.

## Before:

```
class Order {  
    void addNewItem() { new Item(); }  
}
```

## After:

```
class ShoppingCart {  
    void addItem() { new Item(); }  
}
```

# Creator - Overview

## Benefits

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

## Application

- Helps determine the optimal placement of creation logic.
- Ideal for complex systems where proper object creation improves design clarity and maintainability.

# Creator - Before

```
// 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...  
}
```

The `Order` class creates `Item` instances, despite lacking a logical or direct relationship with `Item`. This results in poor encapsulation and weakens the design.

# Creator - After Applying

Assign the responsibility of creating `Item` instances to the `ShoppingCart` class, as it 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

A class representing the overall system or root object handles system events.

**Before:**

```
class UserInterface {  
    void onLoginButtonClick(...) {  
        // directly handle everything  
    }  
}
```

**After:**

```
class UserInterface {  
    void onLoginButtonClick(...) {  
        new LoginController().handleLoginRequest(...);  
    }  
}  
  
class LoginController {  
    // manage login logic  
}
```



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 business logic.
- Simplifies maintenance and enhances scalability.

## Application

- Defines system responses to user actions or events.
- Ensures clear and consistent management of interactions.

# Controller - After Applying

Introduce a `Controller` class to serve 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
    }
}
```

# Understanding Coupling

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.

# Blend Something? Maybe?

```
// 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

Reduce interdependencies between classes.

**Before:**

```
class Order {  
    Payment payment;  
    void processOrder() { payment.processPayment(); }  
}
```

**After:**

```
class Order {  
    PaymentProcessor processor;  
    void processOrder() { processor.processPayment(); }  
}
```

# Low Coupling

- **Benefits:**

- Enhances system flexibility and resilience to changes.
- Simplifies testing and maintenance.

- **Application:**

- Essential for systems expecting changes, minimizing ripple effects from modifications.

# 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 too dependent on the `CoffeeMaker`, causing unpredictable breakfast outcomes.



# Low Coupling - After Applying

- **Impact:** The `Toaster` and `CoffeeMaker` now operate independently, avoiding breakfast chaos, thanks to 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

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

# High Cohesion

A class should have closely related responsibilities.

**Before:**

```
class UserManager {  
    void createUser() { ... }  
    void deleteUser() { ... }  
    void generateReport() { ... } // unrelated  
}
```

**After:**

```
class UserManager {  
    void createUser() { ... }  
    void deleteUser() { ... }  
}  
  
class ReportGenerator {  
    void generateReport() { ... }  
}
```

# High Cohesion

- **Benefits:**

- Simplifies system understanding.
- Improves code management and modification.
- Encourages single responsibility and focused class design.

- **Application:**

- Ensures an organized system with independent, focused components for better design and easier updates.

# High Cohesion - Before Applying

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

- `UserManager` handles both user management and report generation, reducing cohesion and increasing complexity.

# 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` ensures more focused, understandable, and maintainable classes.

# Polymorphism

Let subclasses be treated as their parent class, overriding behavior as needed.

## Before:

```
class AnimalSound {  
    void makeSound(Animal a) {  
        if (a instanceof Dog) { ... } else if (a instanceof Cat) { ... }  
    }  
}
```

## After:

```
abstract class Animal { abstract void makeSound(); }  
class Dog extends Animal { void makeSound() { ... } }  
class Cat extends Animal { void makeSound() { ... } }  
  
class AnimalSound {  
    void makeSound(Animal a) { a.makeSound(); }  
}
```

# Polymorphism

- **Benefits:**

- Increases flexibility and reusability by enabling interchangeable use of different classes.
- Simplifies code by removing multiple conditionals.

- **Application:**

- Ideal for behaviors that vary across classes but share a common interface.



# Polymorphism - Before Applying

```
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

```
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 handles its own sound, removing conditional logic in `AnimalSound` and improving scalability and maintainability.

# Pure Fabrication

Create a “made-up” class to achieve lower coupling/higher cohesion.

**Before:**

```
class Order {  
    void saveOrder() { // direct DB code }  
}
```

**After:**

```
class OrderRepository {  
    void saveOrder(Order o) { // DB code }  
}  
class Order { /* no DB code here */ }
```

# Pure Fabrication

- **Benefits:**

- Improves maintainability, reusability, and separation of concerns.
- Supports better encapsulation.

- **Application:**

- Ideal for behaviors that don't align with domain classes or require independent solutions to design problems.

# Pure Fabrication - Before Applying

```
class Order {  
    // Order related data and methods...  
  
    void saveOrder() {  
        // Direct database access code to save the order  
        // This mixes business logic with data access logic  
    }  
}
```

The `Order` class is managing database operations, straying from its primary responsibility and creating a design that's difficult 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`, as a pure fabrication, handles database operations, ensuring cleaner separation of concerns and a more maintainable design.

# **SOLID**

# **Principles**

# What is SOLID?

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



# **SOLID: Core Concepts**

1. Single Responsibility (SRP)
2. Open/Closed (OCP)
3. Liskov Substitution (LSP)
4. Interface Segregation (ISP)
5. Dependency Inversion (DIP)

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



# SINGLE RESPONSIBILITY PRINCIPLE

Just Because You Can, Doesn't Mean You Should

# **SRP (Single Responsibility Principle)**

**One class = one reason to change.**

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

## Before:

```
class UserManager {  
    void createUser() { ... }  
    void sendEmail(String msg) { ... } // unrelated  
}
```

`UserManager` mixes user creation and email sending, reducing cohesion and increasing error risk.

## After:

```
class UserManager { void createUser() { ... } }  
class EmailService { void sendEmail(String msg) { ... } }
```

Each class now has a single responsibility, improving maintainability and clarity.





# OPEN CLOSED PRINCIPLE

Open Chest Surgery Is Not Needed When Putting On A Coat

## **OCP (Open/Closed Principle)**

Open for extension, closed for modification.

- 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)**

## **How?**

Create a class hierarchy that adds functionality via subclasses or interfaces, avoiding changes to existing code.



## Before:

```
class GraphicsEditor {
    void drawShape(Shape shape) {
        if (shape.type == 1) {
            drawRectangle(shape);
        } else if (shape.type == 2) {
            drawCircle(shape);
        }
        // Adding a new shape requires modifying this method
    }
}
```

Adding a new shape requires modifying `GraphicsEditor`, breaking the open/closed principle.

## After:

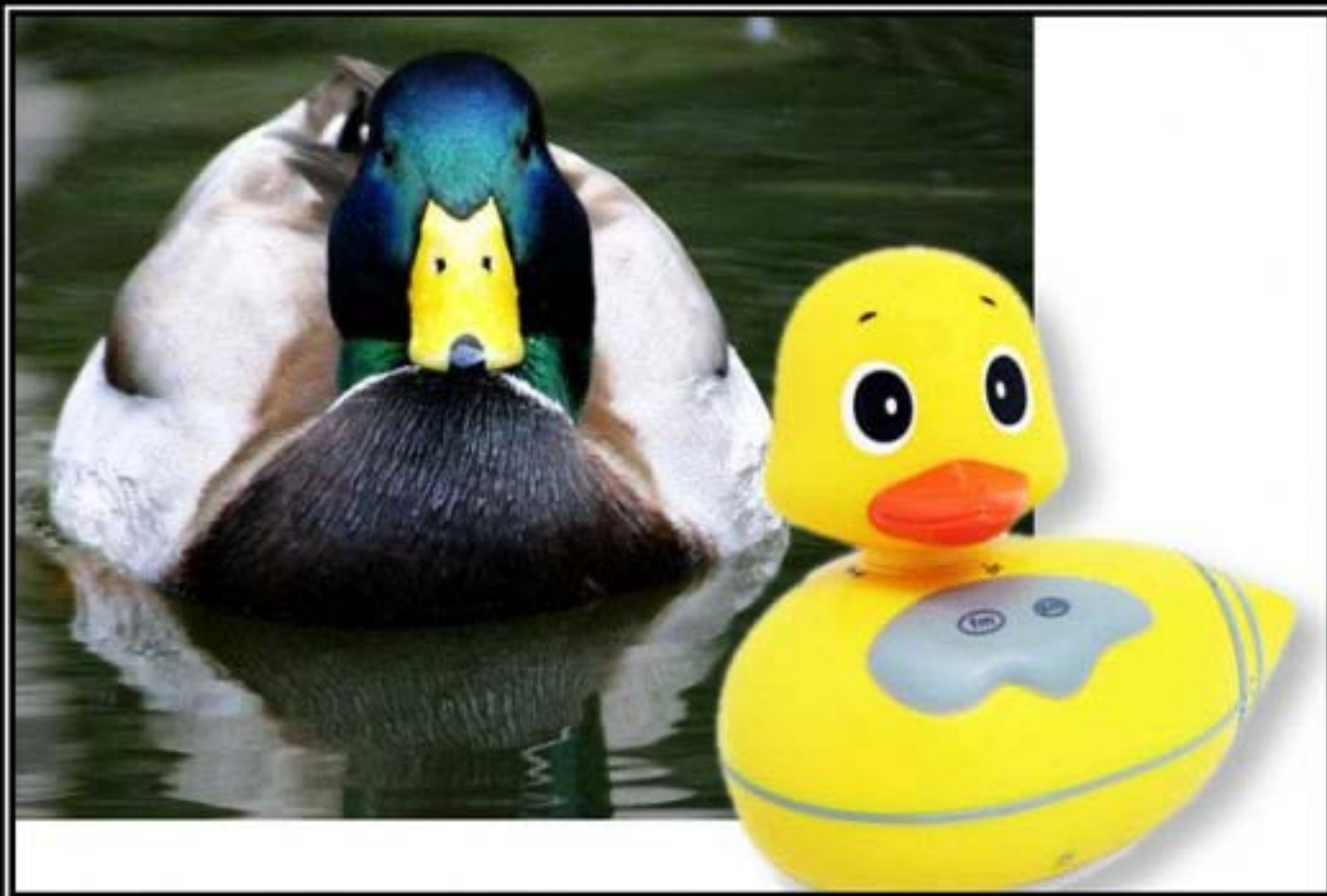
```
abstract class Shape {
    abstract void draw();
}

class Rectangle extends Shape {
    void draw() { /* Draw rectangle */ }
}

class Circle extends Shape {
    void draw() { /* Draw circle */ }
}

class GraphicsEditor {
    void drawShape(Shape shape) {
        shape.draw(); // No modification needed for new shapes
    }
}
```

New shapes can be added without altering `GraphicsEditor`, ensuring extensibility and compliance with the open/closed principle.



# LISKOV SUBSTITUTION PRINCIPLE

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

# **LSP (Liskov Substitution Principle)**

Subclasses must be substitutable for their base classes.

- Ensures a subclass can replace its parent class without unexpected behavior.

## Before:

```
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 as a `Bird` can cause errors due to its overridden `fly` method.

## After:

```
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 substitutable for `Bird`, ensuring compliance with the Liskov Substitution Principle.



# INTERFACE SEGREGATION PRINCIPLE

You Want Me To Plug This In, Where?

# ISP (Interface Segregation Principle)

Don't force classes to implement methods they don't use.

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

## Before:

```
interface Worker {  
    void work();  
    void eat();  
    void takeBreak();  
}  
  
class Robot implements Worker {  
    void work() { /*...*/ }  
    void eat() { /* Robots don't eat */ }  
    void takeBreak() { /* Robots don't take breaks */ }  
}
```

`Robot` is forced to implement irrelevant methods like `eat` and `takeBreak`.

## After:

```
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` implements the `Worker` interface, while `Human` uses the extended `HumanWorker` interface, improving flexibility and maintainability.





# Dependency Inversion Principle

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

# **DIP (Dependency Inversion Principle)**

Depend on abstractions, not concretions.

- 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.

## Before:

```
class OrderProcessor {  
    MySQLDatabase database; // Directly dependent on a specific  
    database implementation  
  
    void processOrder(Order order) {  
        database.save(order); // Tightly coupled to  
        MySQLDatabase  
    }  
}
```

`OrderProcessor` depends on `MySQLDatabase`, making database changes and independent testing difficult.

```
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 relies on a Database abstraction, improving flexibility and simplifying modifications or testing.

**See you in the  
lab!**